WRDC-TR-89-1139

THE INTERACTIVE ADA WORKSTATION

General Electric Company
Corporate Research and Development
P. O. Box 8
Schenectady, NY 12301

May 1990

Final Report for Period : 30 July 1985 - 30 May 1989

AVIONICS LABORATORY
WRIGHT RESEARCH DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

WHEN GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA ARE USED FOR ANY PURPOSE OTHER THAN IN CONNECTION WITH A DEFINITELY GOVERNMENT-RELATED PROCUREMENT, THE UNITED STATES GOVERNMENT INCURS NO RESPONSIBILITY OR ANY OBLIGATION WHATSOEVER. THE FACT THAT THE GOVERNMENT MAY HAVE FORMULATED OR IN ANY WAY SUPPLIED THE SAID DRAWINGS, SPECIFICATIONS, OR OTHER DATA, IS NOT TO BE REGARDED BY IMPLICATION, OR OTHERWISE IN ANY MANNER CONSTRUED, AS LICENSING THE HOLDER, OR ANY OTHER PERSON OR CORPORATION; OR AS CONVEYING ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY IN ANY WAY BE RELATED THERETO.

THIS REPORT HAS BEEN REVIEWED BY THE OFFICE OF PUBLIC AFFAIRS (ASD/PA) AND IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS IT WILL BE AVAILABLE TO THE GENERAL PUBLIC INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

Otheus Jackson

Otheus Jackson
Project Engineer
Software Concepts Group

Donna M Morris

Donna M. Morris, Chief
Avionics Logistics Branch
System avionics Division

FOR THE COMMANDER

Charles H Krueger

CHARLES H. KRUEGER
Director
System Avionics Division
Avionics Laboratory

IF YOUR ADDRESS HAS CHANGED, IF YOU WISH TO BE REMOVED FROM OUR MAILING LIST, OR IF THE ADDRESSEE IS NO LONGER EMPLOYED BY YOUR ORGANIZATION PLEASE NOTIFY WRDC/AAAF-3, WRIGHT-PATTERSON AFB, OH 45433-6543 TO HELP MAINTAIN A CURRENT MAILING LIST.

COPIES OF THIS REPORT SHOULD NOT BE RETURNED UNLESS RETURN IS REQUIRED BY SECURITY CONSIDERATIONS, CONTRACTUAL OBLIGATIONS, OR NOTICE ON A SPECIFIC DOCUMENT.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): WRDC-TR-89-1139
6a. NAME OF PERFORMING ORGANIZATION: General Electric Company Corporate Research and Development; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: WRDC/AAAF
6c. ADDRESS (City, State, and ZIP Code): P.O. Box 8 Schenectady, NY 12301	7b. ADDRESS (City, State, and ZIP Code): WPAFB, OH 45433-6543
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Same; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F36615-85-C-1755
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 62204F; PROJECT NO. 2003; TASK NO 02; WORK UNIT ACCESSION NO. 77

11. TITLE (Include Security Classification)

The Interactive Ada Workstation

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Jul 85 TO May 89	May 1990	115

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Ada, Expert System, CASE, Automatic Programming, Rapid Prototyping.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Interactive Ada Workstation (IAW) major concern is with decreasing the cost and development time of future systems implemented with Ada programming language. During the design phase, developing software with Ada takes longer and cost more than with conventional languages. The objective of the IAW is to demonstrate significantly improved Ada programming productivity through the use of rapid prototyping techniques.

This software was developed with the LISP language for a Symbolics machine. The work was centered around graphical techniques and the mapping of abstract design to the Ada language. This project developed four main graphical editors from Buhr Diagrams, State Machine Diagrams, Decision Tables, and Truth Tables. Each editor is capable of generating Ada code. The Buhr Representation and Ada Translator (BRAT) Editor is used for the specification of the hierarchical structure of Ada program elements, and the calling relationship between the elements. This data is used to produced Ada specifications. The State Machine Editor (SME) is used to model the program behavior base on time intervals or change (continued)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: OTHEUS JACKSON	22b. TELEPHONE (Include Area Code): (513) 255-6548; 22c. OFFICE SYMBOL: WRDC/AAAF-3

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

BLOCK 19 Continued

of logical inputs. The Decision Table Editor (DTE) is used to describe the relationship of systems or subsystems in terms of conditions, events, or rules. The Truth Table Editor permits the user to describe the behavior of a BRAT object in terms of its input and output. There are other editors which are subsets of the main editors. In order to integrate the four editors and permit concurrent updates, the Abstract Semantic Model was developed. Conceptually the semantic model is viewed as data flow diagram, with each node representing a computation and the arcs between the nodes representing the data that passes between the computations. Other accomplishments include an interpreter, a parser, and an incremental semantic checking (compile-as-you-type). The IAW is one of the first Computer Aided Software Engineering (CASE) tools to focus solely on the Ada Language. It is also suitable for developing and maintaining very large scale software systems. The concept and technology of the IAW is part of a commercially supported product, TEAMWORK/ADA from Cadre Technologies.

Table of Contents

List of Figures

1

Executive Summary

The fundamental objective of the Interactive Ada Workstation (IAW) research contract (F36615-85-C-1755) was to significantly improve Ada programming productivity through the use of interactive software technology, and demonstrate this capability through a series of prototypes. Much was accomplished directly under the auspices of the contract to achieve this objective, and as a by-product of the contract, the research matured the IAW technology to the point where its continued development into commercially available tools became practical.

As a result of the successful research performed under the IAW contract, GE signed a joint development agreement with Cadre Technologies of Providence, R.I. to build a series of tools based upon this work. Today, the first tools based upon this technology, the Ada Structure Graph (ASG) editor and Ada Source Code Builder (ASB), are available from Cadre as commercially supported products, and other major elements of the IAW are being readied for commercialization. These tools, and the Teamwork/Ada environment in which they operate, have already been specified for use in several major US government programs, including the Space Station Environment (Lockheed), the Joint Interactive Avionics Working Group (for LHX Helicopter, ATF and ATA avionics) and the Stars program (Boeing).

Given the technical difficulties that were encountered during the research, and the time and resources that it took to overcome them, it is highly unlikely that these commercial offerings would be available today had it not been for the active support of the Air Force and the IAW contract.

1.1 Technical Accomplishments

A number of significant technical milestones were achieved during this contract. Among these are:

- The formalization of Buhr diagrams, state machine diagrams, decision tables, and truth tables so that Ada code could be generated from the diagrams.
- The demonstration that these graphic tools could be integrated in such a way that the behavioral specification tools (such as the state machine and decision table editors) could be entered from the Buhr structure editor to provide behavioral descriptions for components that had been structurally defined by the Buhr diagram. Integrated Ada code generation was also demonstrated.
- The demonstration of the feasibility of instantaneous syntax checking of the complete Ada language without constraining the manner in which the user performs his edits.
- The demonstration of the feasibility of incremental semantic checking (compile-as-you-type) with nearly real-time feedback to the user for a substantial subset of Ada.
- The demonstration of the feasibility of integrating an interpreter with the incremental editor so that programs that are in the process of being interpreted may be modified without restarting the interpretation.
- The development of an abstract language model for supporting incremental semantic analysis that can be extended to cover the entire Ada language. (This extension was, in fact, carried out under GE funding after the IAW stop work order was issued.)
- A demonstration that help systems such as data structure selectors, search/sort algorithm selectors, and smart librarians could be integrated with these tool sets.

1.2 Summary of Work

The IAW contract called for the development of a series of prototypes written in Lisp on Symbolics workstations. This platform was chosen to facilitate the rapid development of these prototypes. At the start of the contract, rudimentary technology was already in place on this platform for capturing individual graphic representations of a design and producing an outline of the Ada code necessary to implement that portion of the design shown in the diagram. This capability was demonstrated in the P0 prototype delivered at the start of the contract.

Two major parallel thrusts were embarked upon: one to extend and integrate the various graphic representations to make them more complete, and to generate an integrated Ada code image; and the second, to develop a text editor that would compile Ada code while it is being entered into an interpretable form, and provide an interpreter for this form. The ultimate goal was to integrate these two efforts into a system that would keep the design (as represented in the graphics) consistent with the text throughout the design cycle, even in the face of subsequent changes to the code. In addition to these main thrusts, peripheral thrusts in help systems, expert advisors and design checking were also initiated.

The graphic thrust matured quite quickly, as evidenced by the P1 prototype's integration of graphic specifications of software structure and behavior. The language effort, on the other hand, turned out to be a more formidable task. The original intent was that the prototype language tool, which was intended to demonstrate the editor, syntax and semantic checking, and the interpreter for a simple expression language, would be delivered as part of the P1 prototype. In fact, this prototype language tool took longer than expected to develop, and was not delivered until the P2 prototype, which also included an editor and parser (syntax checker) for the entire Ada language.

At this point, the bulk of the effort on the project became devoted to completing the language coverage: developing and refining a strategy for semantic analysis (compilation) and applying the strategy to increasing segments of the Ada language. Prototype P3 contained the first semantic analysis capability, and P4 contained extended language coverage and an interpreter.

While the contract initially called for a series of seven prototypes to be developed at 6-month intervals, subsequent alterations in the funded amount, periodic suspensions of funding, and, ultimately, the receipt of a stop work order resulted in the actual delivery of only five completed prototypes under the contract. A sixth prototype, developed with GE funding during one of the suspensions of Air Force funding, was provided to the Air Force under the terms of a memorandum of understanding regarding proprietary data. Work was in progress toward a seventh and final prototype when the stop work order was received.

2

Introduction

Summary

Corporate Research and Development of the General Electric Company submits this Final Technical Report for the Interactive Ada Workstation (IAW). This report constitutes fulfillment of CDRL Sequence Number 9, CLIN00001, for contract F33615-85-C-1755 awarded in July 1985 by the Department of the Air Force, Aeronautical Systems Division, Wright-Patterson Air Force Base, to the General Electric Company.

This chapter introduces the project and provides an overview of its technical evolution beginning with the P0 prototype, continuing through the P1, P2, P3 and P4 prototypes, and the GE-developed L5 prototype. It also summarizes the state of the project at the time that the stop work order was issued.

2.1 Overview of Technical Evolution

The IAW contract called for the development of a series of prototypes written in Lisp on Symbolics workstations. At the start of the contract, rudimentary technology was already in place on this platform for capturing individual graphic representations of a design and producing an outline of the Ada code necessary to implement that portion of the design shown in the diagram. This capability was demonstrated in the P0 prototype delivered at the start of the contract.

Two major parallel research thrusts were embarked upon: one to extend and integrate the various graphic representations to make them more complete, and to generate an integrated Ada code image; and the second, to develop a text editor that would compile Ada code while it is being entered into an interpretable form, and provide an interpreter for this form. The ultimate goal was to integrate these two efforts into a system that would keep the design (as represented in the graphics) consistent with the text throughout the design cycle, even in the face of subsequent changes to the code. In addition to these main thrusts, peripheral thrusts in help systems, expert advisors and design checking were also initiated.

The graphic thrust matured quite quickly, as evidenced by the P1 prototype's integration of graphic specifications of software structure and behavior. The language effort, on the other hand, turned out to be a more formidable task. The original intent was that the prototype language tool, which was intended to demonstrate the editor, syntax and semantic checking, and the interpreter for a simple expression language, would be delivered as part of the P1 prototype. In fact, this prototype language tool took longer than expected to develop, and was not delivered until the P2 prototype, which also included an editor and parser (syntax checker) for the entire Ada language.

At this point the bulk of the effort on the project became devoted to completing the language coverage: developing and refining a strategy for semantic analysis (compilation) and applying the strategy to increasing segments of the Ada language. Prototype P3 contained the first semantic analysis capability, and P4 contained extended language coverage and an interpreter.

While the contract initially called for a series of seven prototypes to be developed, at 6-month intervals, subsequent alterations in the funded amount, periodic suspensions of funding, and, ultimately, the receipt of a stop work order resulted in the actual delivery of only five completed prototypes under the contract. A sixth prototype, developed with GE funding during one of the suspensions of Air Force funding, was provided to the Air Force under the terms of a memorandum of understanding regarding proprietary data. Work was in progress toward a seventh and final prototype when the stop work order was received.

The following sections describe, in more detail, the five prototypes (referred to as P0 through P4) delivered under the contract, the additional prototype developed by GE during a suspension in the funding (the prototype referred to as L5), and the work that was in progress at the time that the stop work order was received.

2.1.1 P0 Prototype

The P0 prototype reflected work done at the GE prior to the contract. This previous work was assembled and delivered to the Air Force under the terms of the contract. This prototype consisted of several isolated graphics editors, including:

1. the Buhr Representation and Ada Translator Editor (BRAT),
2. the State Machine Editor (SME),
3. the Decision Table Editor (DTE)

Each of these editors had code generation capability. In addition, the prototype included a Rewrite Rule Laboratory (RRL), which could be used as the basis for a formal verification system, and a Help System.

2.1.1.1 The BRAT Editor

The BRAT (Buhr Representation and Ada Translator) Editor was designed to facilitate the specification of the hierarchical structure of Ada program elements, and the calling relationships between the elements. In addition to basic structure, the editor captured the parameterization required of subprograms, and the type specifications of variables and constants. Collectively, this is enough information to produce the Ada specifications of all program units shown in the diagrams, and an outline of the body for each program unit.

2.1.1.2 The State Machine Editor

The State Machine Editor (SME) allows the user to create, modify and simulate state machines. State machines provide a method for developing a formal specification of program behavior. The state diagram which is constructed with the SME is one method for representing that solution.

Any problem that can be considered as being in one of a finite number of "states" at some point in time, and that changes state based upon a boolean combination of logical inputs, can be described using this editor.

2.1.1.3 The Decision Table Editor

The Decision Table Editor (DTE) allows the user to design systems or subsystems by describing them in terms of relationships between conditions and events, rules, and relationships between rules called tables. The designer specifies a number of input conditions, and a series of rules indicating which actions should be taken when the input conditions for the rule are satisfied. The editor can then automatically generate code that implements the rule-based decisions.

2.1.2 P1 Prototype

The P1 prototype represented the first major segment of work produced under the contract. While the original intent was that the P1 prototype was to include both graphics and language capability, the P1 language capability took longer than anticipated to develop. The P1 language tools were actually delivered at the same time as the P2 prototype.

2.1.2.1 P1 Graphics

The P1 prototype integrated four major graphical editors for the IAW. These included:

1. the Buhr Editor (BRAT),
2. the State Machine Editor (SME),
3. the Truth Table Editor (TTE), and
4. the Decision Table Editor (DTE)

The BRAT Editor was significantly improved to coordinate the operation of the other editors and expert systems. In addition, a new subeditor, the Black-Box Editor, was added to BRAT.

The Black-Box Editor (BBE), is used to specify the external interface of each object in a hierarchy. The Black-Box Editor describes an object at the external socket level. User options include adding input, output, and input/output parameters. Users are prompted to enter information concerning type, source, rate and initial value. When the description is completed, the information is automatically translated to a BRAT description of the object's externally visible sockets.

Once an object is defined in the BRAT Editor, it can be further specified and described by the other three editors. While all editors participate in the capture of a design, the BRAT Editor handles all requests for code generation. In the P0 prototype, the user was required to manually merge code from each of the three editors. The P1 release eliminated the need to manually merge the code from the various editors.

Two expert systems, or designer's assistants, were made available through BRAT. These included:

1. the Data Structure Selector
2. the Search/Sort Selector

The Data Structure Selector guides the user to an appropriate implementation for an abstract data type, based on the structure and contents of the information, and the operations that the user expects to be frequently performed. Additional constraints, such as memory availability, were also considered.

The Search/Sort Selection design tool helped the user select the appropriate search or sort algorithm for a data structure. It asked a sequence of relevant questions, and then recommended the most efficient procedure based upon user input.

2.1.2.1.1 The Truth Table Editor

The Truth Table Editor (TTE) allows the user to describe the behavior of a BRAT object in terms of its inputs and outputs (as derived from the BBE in BRAT). This behavior description takes the form of a boolean equation which is derived from the user's specification of output values for particular sets of input values.

2.1.2.2 The P1 Language

The P1 Language Prototype, which was delivered at the same time as the P2 prototype, was comprised of:

- an editor
- a parser
- a semantic analyzer, and
- an interpreter

The primary reason for producing the P1 language prototype was to demonstrate the intended user interface, and show how the final system would work using a very small and simplified language subset as an example. We anticipated that this approach would allow the prototype to be relatively quickly produced and would generate the necessary feedback about how the system presented itself to the user.

Unfortunately, even using this simplified language subset, the prototype took longer than anticipated to develop, and was, in fact, delivered at the same time as the P2 prototype.

The P1 prototype language was a small Ada subset, adequate to generate meaningful programs. Block statements were the compilation unit. A block contained variable declarations and a sequence of statements. Variables were declared as type integer or type boolean, and a variable could be assigned an initial value. The sequence of statements consisted of simple if statements, loop statements, and assignment statements which were executed in succession.

While the P1 language processing approach was incrementally effective and relatively fast, it was not capable of handling the more complicated parsing requirements of the full Ada language. An example of the simplified parsing is shown in figure 2-1.

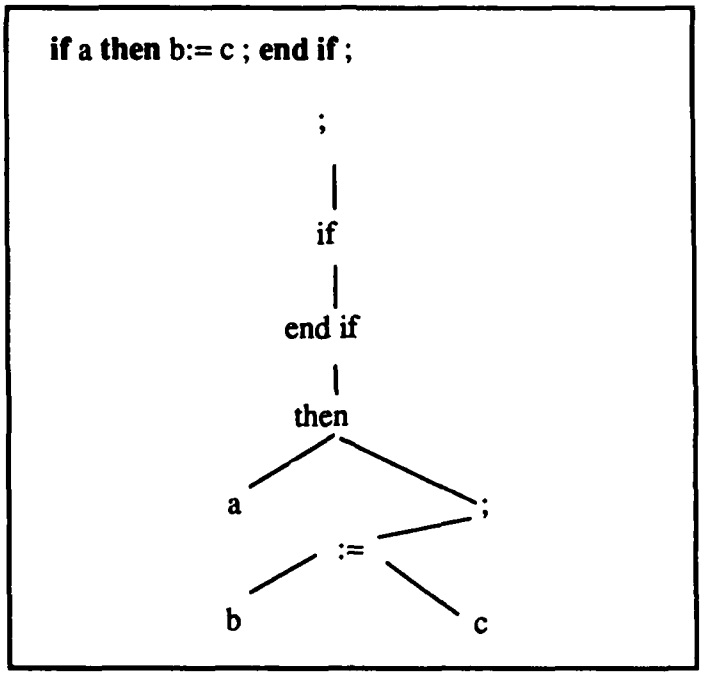


Figure 2-1. P1 Parsing Example

2.1.3 P2 Prototype

In examining the P1 prototype, we realized that the changes required to extend the P1 prototype language coverage to the entire Ada language would be so extensive that it would be easier to abandon the prototype and begin again. This was largely due to the P1 prototype's dependence on assumptions concerning precedence, associativity, and keywords having a single meaning which were not valid for the full Ada language.

The P2 prototype included:

1. centralized window and process management
2. elements of a full language system including
   - text editor with reasonable range of functions
   - more sophisticated parsing strategy with full Ada language syntax coverage
3. a demonstration of batch transfer capability between the proposed semantic data structure and BRAT graphics data structure
4. VHDL code generation (from graphics)

The parser used in this prototype and subsequent prototypes was based on work done by Ghezzi and Mandrioli. Extensions were developed to allow it to handle the full Ada language. The resulting parser uses full bottom-up parsing techniques, and separates language dependencies into parse tables that are used by a generic parsing kernel. Figure 2-2 shows an example of an Ada statement parsed with this technique.

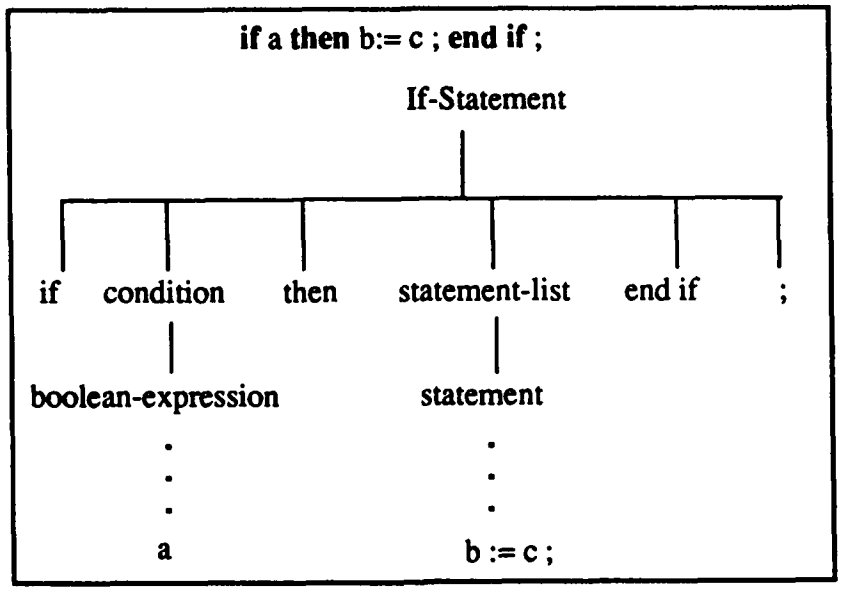


Figure 2-2. P2 Parsing example

2.1.4 P3 and P4 Prototypes

The P3 and P4 prototypes included:

1. A refined text editor
2. A separate lexical analyzer
3. A refined parser, with improved reuse of nodes in the parse tree
4. Semantic analysis for a substantial portion of Ada and an error message window with text interaction.
5. Increased language coverage including:
   - all predefined types except string and duration
   - user-defined types
     - enumerations
     - subtypes

- nonvariant records

- implicit operators and user-defined operators
- attributes for simple types
- predefined and user-defined exceptions and raise statements
- pragmas
- qualified expressions

6. Increased semantics performance including:
   - static and dynamic expressions
     - compile time evaluation of static expressions
     - dynamic expressions prepared for interpretation
   - statements and executable units
     - analyzed and prepared for interpretation
   - range checking
     - case statement support
7. Batch transfer between actual semantic data structure and Buhr graphics data structure
8. A redesigned structure editor, developed from the VHDL program, which replaced the Buhr Editor
9. A unified data structure for graphic editors
10. An interpreter and inspector for an Ada subset
11. A Smart Librarian

2.1.4.1 Lexical Analysis

In examining the P2 prototype, we found that some incremental changes relating to strings, comments and character literals were not being properly handled. In the P2 prototype, the lexical analysis (which is responsible for identifying the boundaries of tokens, including strings, character literals and comments) was split between the editor and the parser. This situation made the correction of these problems very difficult and error prone. In consequence, we decided to gather the lexical analysis functionality in one place for the P3 prototype.

2.1.4.2 Incremental Parsing and Node Reuse

In the style of parser used in P2 and beyond, the parser constructs a tree (figure 2-3) that relates the terminal tokens (visible text on the screen) to the grammar rules that are used to recognize constructs in the language. In incremental parsing, generally one of the nodes has been changed, and the recognition rules must be applied again to determine what has changed.

An issue in incremental parser design is the reuse of the nodes in the parse tree. This is equivalent to recognizing that the rule used to recognize a construct before the change is the same rule after. When just performing syntax analysis, node reuse is primarily an efficiency issue; the sooner the parser can recognize that nothing has changed, the sooner the reparse can end.

When the parser is being used in conjunction with semantic analysis, the situation changes considerably. Just as the parser plays the role of a recognizer of constructs in the language, it also serves as a recognizer of significant program elements: the declaration of a package or subprogram, for example. If the parse tree node that is associated with recognizing a program element is not reused (i.e., is destroyed and rebuilt), then the abstract representation of this program element will also be destroyed and rebuilt. If no other editors are present in the system, this will simply cause additional semantic reanalysis. However, if other editors have associated information with this abstract representation (graphic placement and layout data from a structure editor, for example), this information will be lost when the abstract representation is destroyed and rebuilt. For this reason, considerable effort was expended to improve node reuse in the P3 and P4 prototypes.

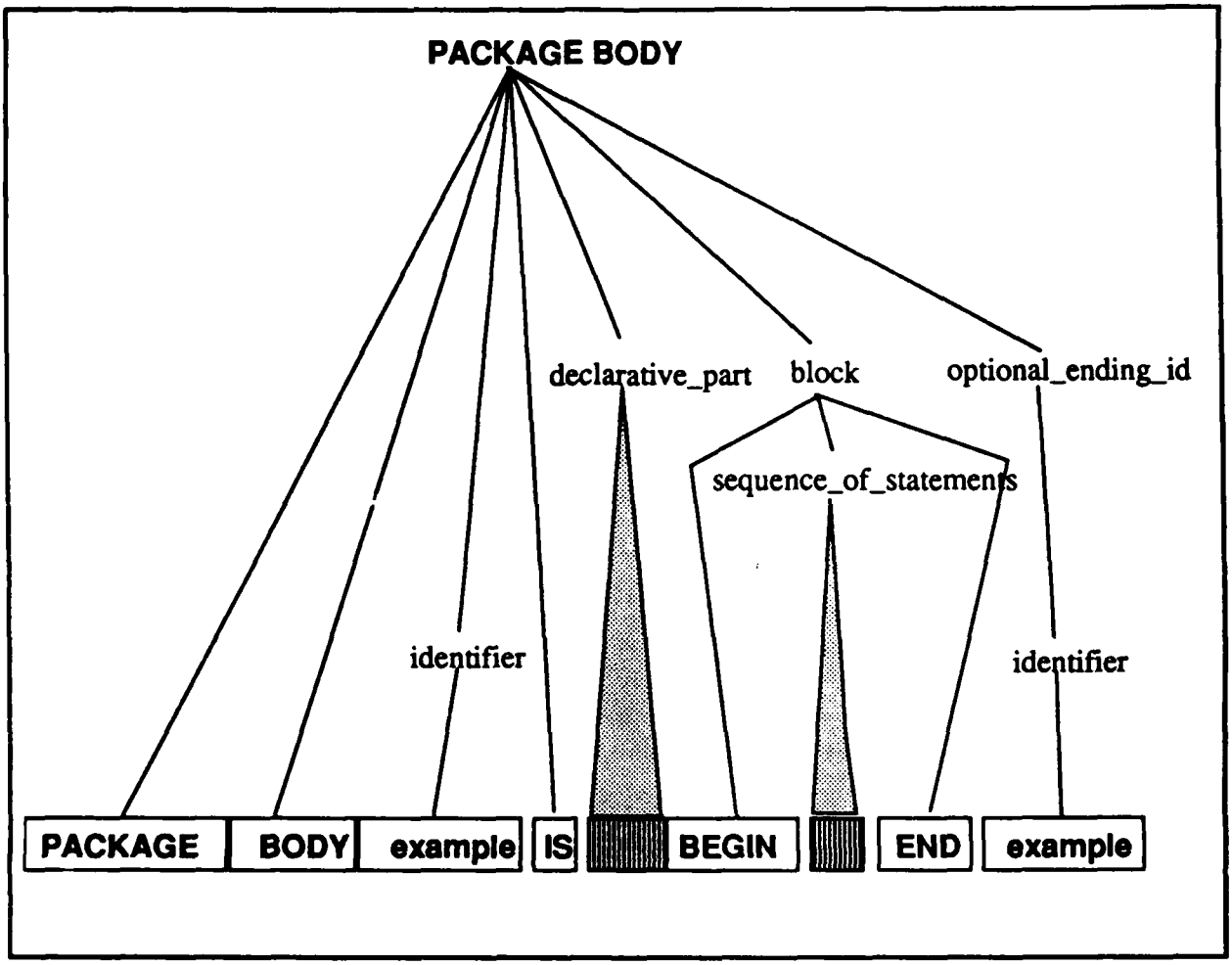


Figure 2-3. Simplified Parse Tree Example

2.2 GE Developed Prototype L5

After the delivery of the P4 prototype, the contract funding was suspended for a period of time. Then, the research was continued with GE funding. The resulting L5 prototype was made available to the Air Force under the terms of a memorandum of understanding regarding GE proprietary funding.

Work on the L5 prototype focused primarily on completion of the language coverage, and on the generalization of the abstract model for use as a common representation for both the structure editor and the language view.

2.3 Interrupted Work: Prototype P6

When funding was restored to the program, work began on a final deliverable, the P6 prototype. Approximately halfway through the planned effort on this prototype, a final stop work order was issued. When the order was issued, coding had just begun. The prototype never progressed to the stage where the results could be demonstrated.

The focal point of the P6 prototype was concurrent update between the language representation and the graphic structure editor. An overview of the relationship between the graphic editor and the abstract model is given in chapter 6. Some rearchitecting of the abstract model processing, attribute processing, and semantic error reporting was done to make the operation of the parser independent of the abstract model, and to separate semantic error reporting from the parser activity. These changes are described in more detail in chapter 7.

3

Architectural Changes

Summary

This chapter describes the design evolution and architectural changes from prototype to prototype.

3.1 Design Concepts

The original concept for the final prototype of the IAW is shown in figure 3-1. In this view, the IFORM Database contains a view of the design that is shared between all of the editors and tools. Each editor has, in addition, some information that is specific to itself. What was not appreciated in this initial view is that the lexical and syntactic information is not really part of the shared view, but is really language editor specific information, just as graphic placement and layout information is specific for a given graphic editor.

In this original design concept, we envisioned that the editor specific information (aside from the language information) would simply be a data structure stored separately from the abstract model. As work progressed through the various prototypes, we recognized that some processing was required to keep these auxiliary data structures consistent with the abstract model the same as the parse tree is kept current with the abstract model.

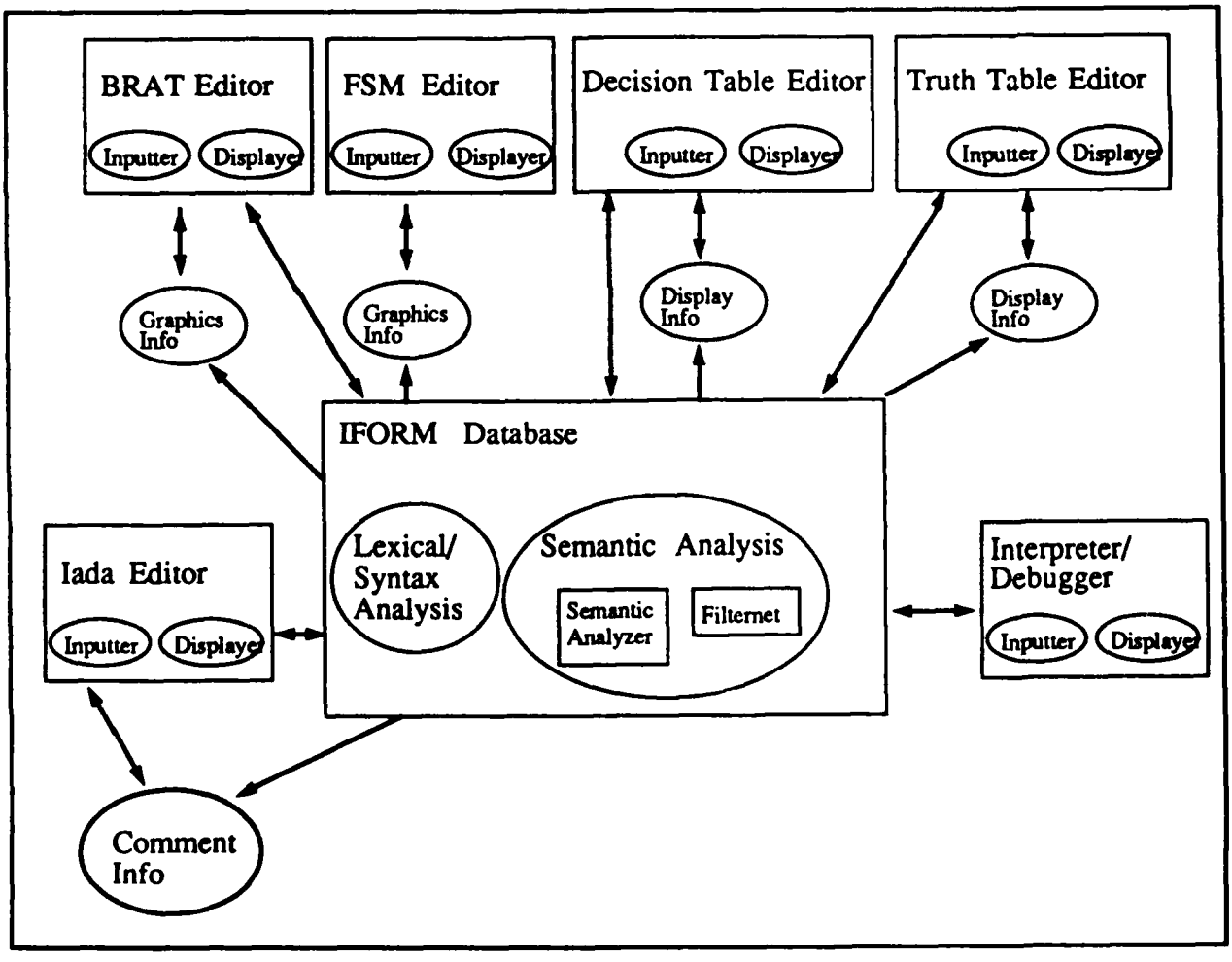


Figure 3-1. Original Design Concept

From another perspective, it is convenient from an editor's perspective to have a view of the abstract model that has its view-specific information smoothly integrated in with the rest of the abstract model. This requires some separate processing to relate the abstract model and the auxiliary data. The language editor views only the auxiliary data and only indirectly interacts with the abstract model. In the case of the graphics structure editor, the relationship is more obvious, although not completely so (see chapter 6 for more details). This leads to a revised design concept as shown in figure 3-2.

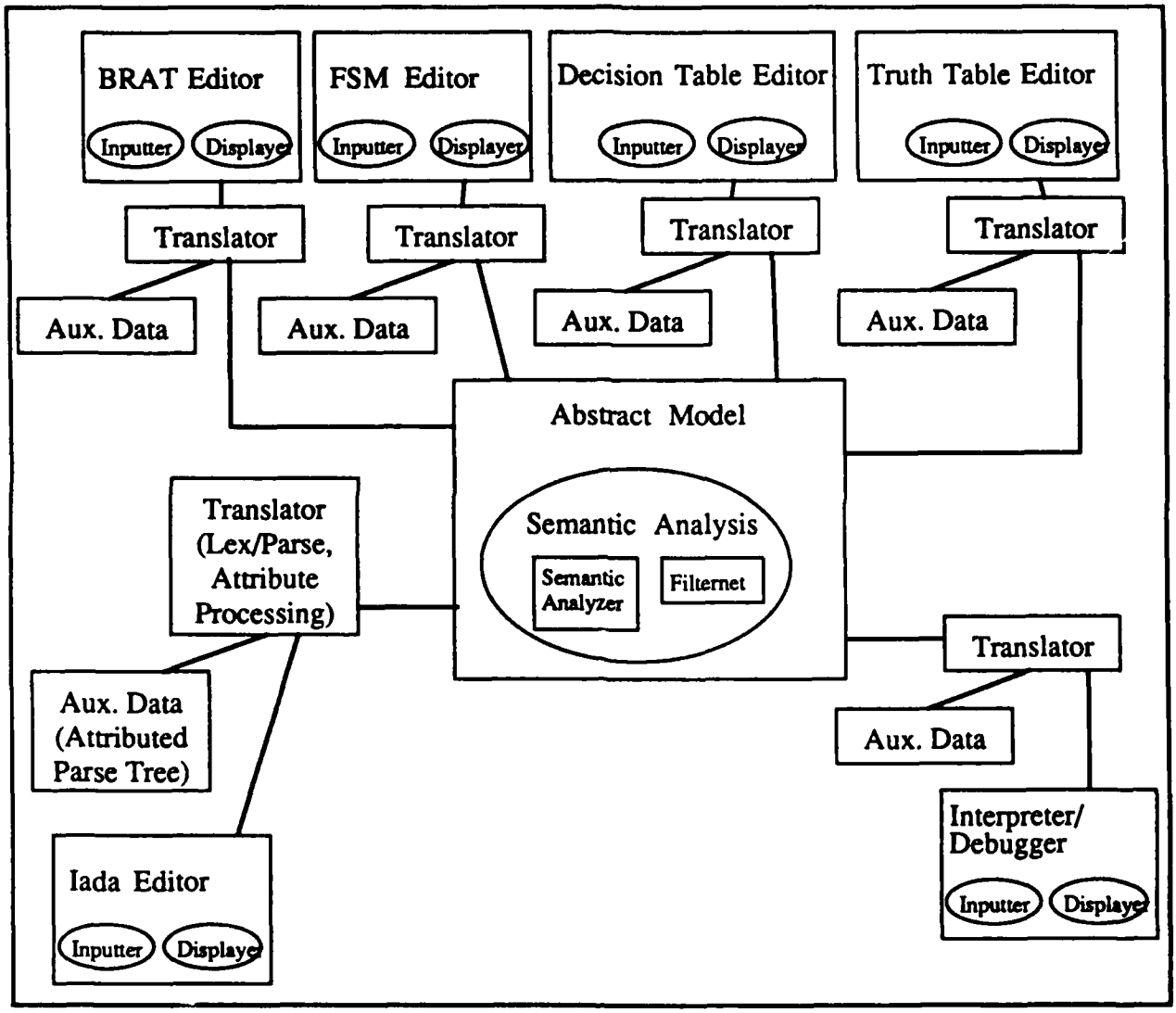


Figure 3-2. Current Design Concept

3.2 P0 Prototype

A block diagram of the P0 prototype is shown in figure 3-3. In this prototype, each editor was completely independent of the others, and each had its own database. While code could be generated from each editor, the code from the various editors had to be manually merged and then added to for a complete program. This made it difficult to go back and change the graphics since the subsequent editing of the code was labor intensive and error prone.

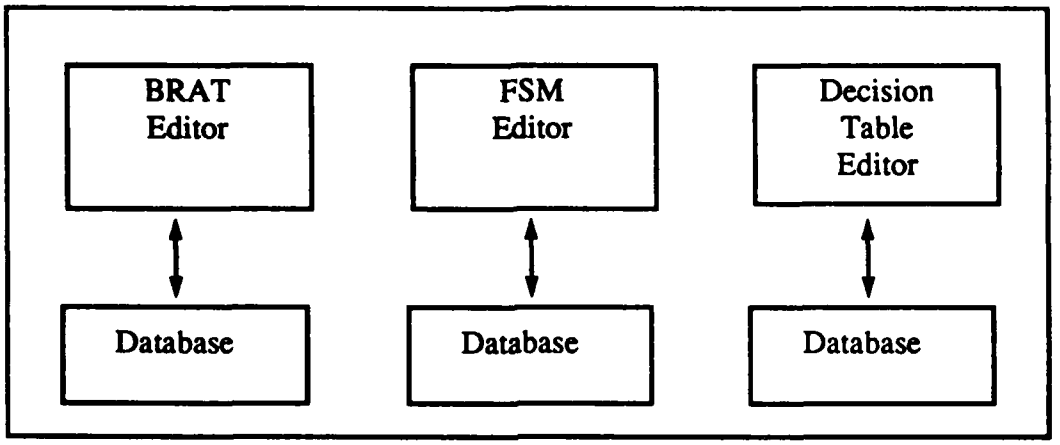


Figure 3-3. P0 Prototype

3.3 P1 Prototype

With the P1 prototype (figure 3-4), the operation of the graphic editors was coordinated. The state machine, decision table and truth table editors could be entered from the BRAT editor, so that the user could easily move from structure to behavioral specification and back again. Although each editor still maintained its own database, code generation was also coordinated by the BRAT editor, eliminating the manual editing step of combining the code together from the various editors. However, the user was still left with the problem of integrating manually generated code.

The language editor, database and interpreter are shown as part of this prototype, even though they were actually delivered at the same time as the P2 prototype.

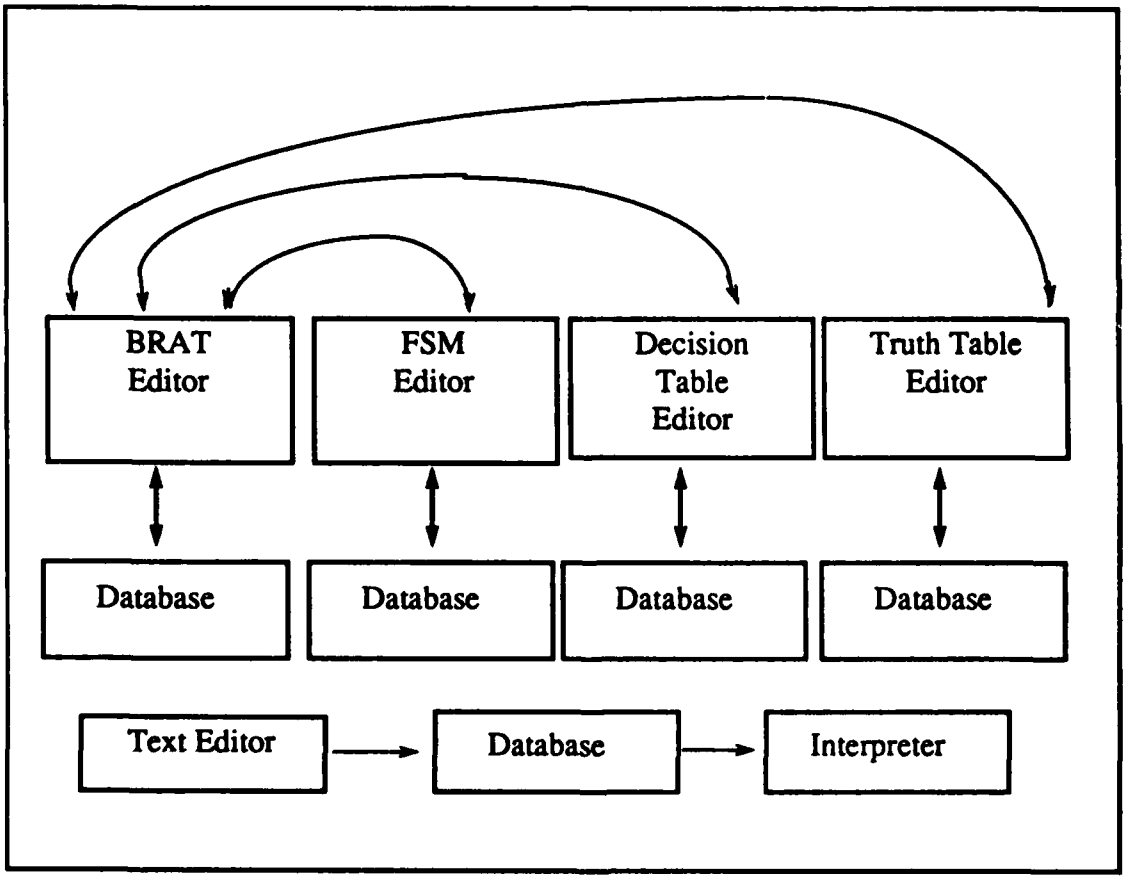


Figure 3-4. P1 Prototype

3.4 P2 Prototype

Architecturally, there was not much change between the P1 and P2 prototypes (figure 3-5). The major change was in the language editors, in which the demonstration editor/displayer of the P1 was replaced with a completely new editor, and the parser portion of the new database replaced the P1 database. The construction of the semantic portion of this database was not completed for the P2 prototype, so the translation capability between the language database and the BRAT editor was from a proposed data structure rather than an actual one. There was no interpreter in the P2 prototype.

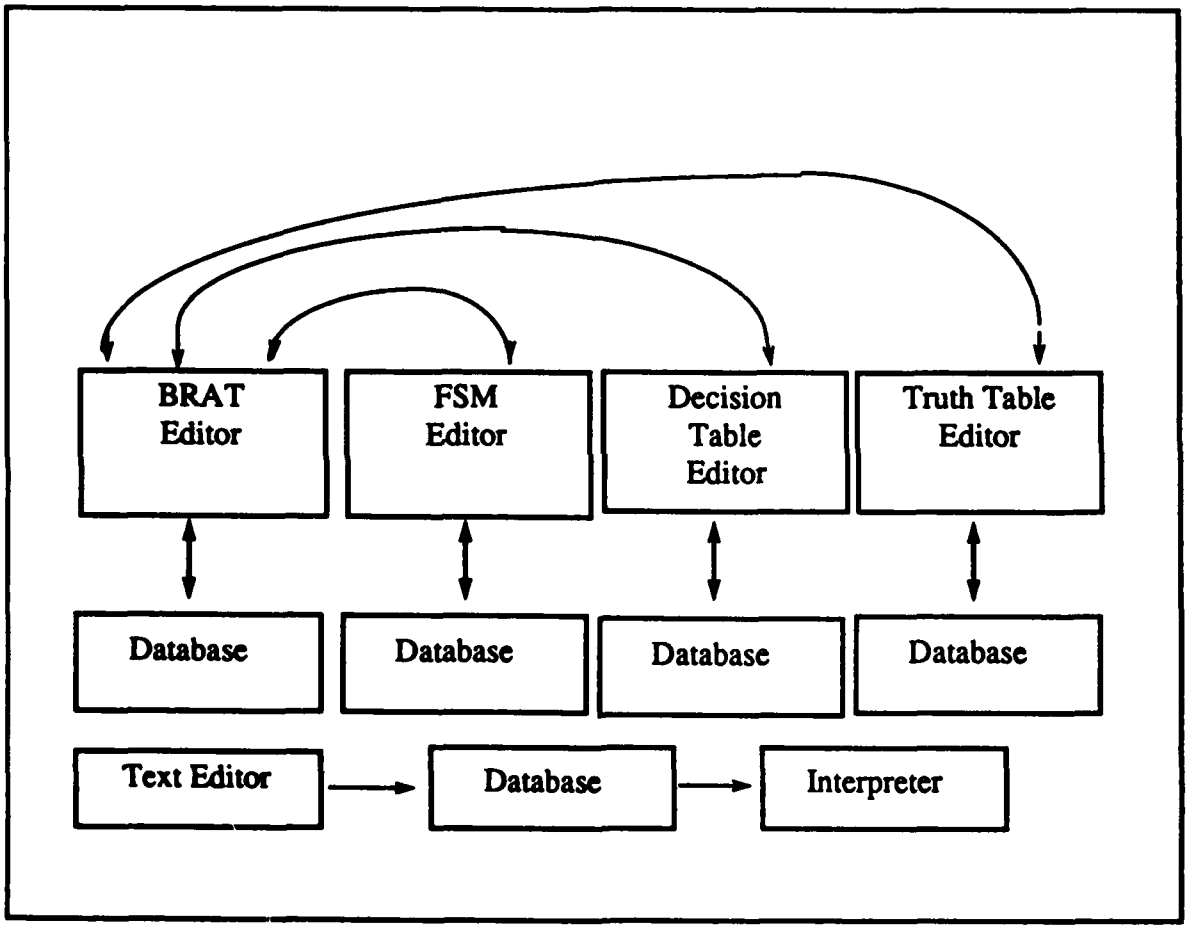


Figure 3-5. P2 Prototype

3.5 P3, P4 and L5 Prototypes

A functional block diagram of the P3, P4 and L5 prototypes is shown in figure 3-6. Note that the graphics editors are now using a unified graphics database (the truth table was not carried over into this prototype). The batch transfer capability from the abstract model to the graphics database only transfers declarative information about the program structure and references: details of executable code are not transferred in these prototypes.

Note that the displayer for the language view plays a dual role of displaying the text and the semantic error messages.

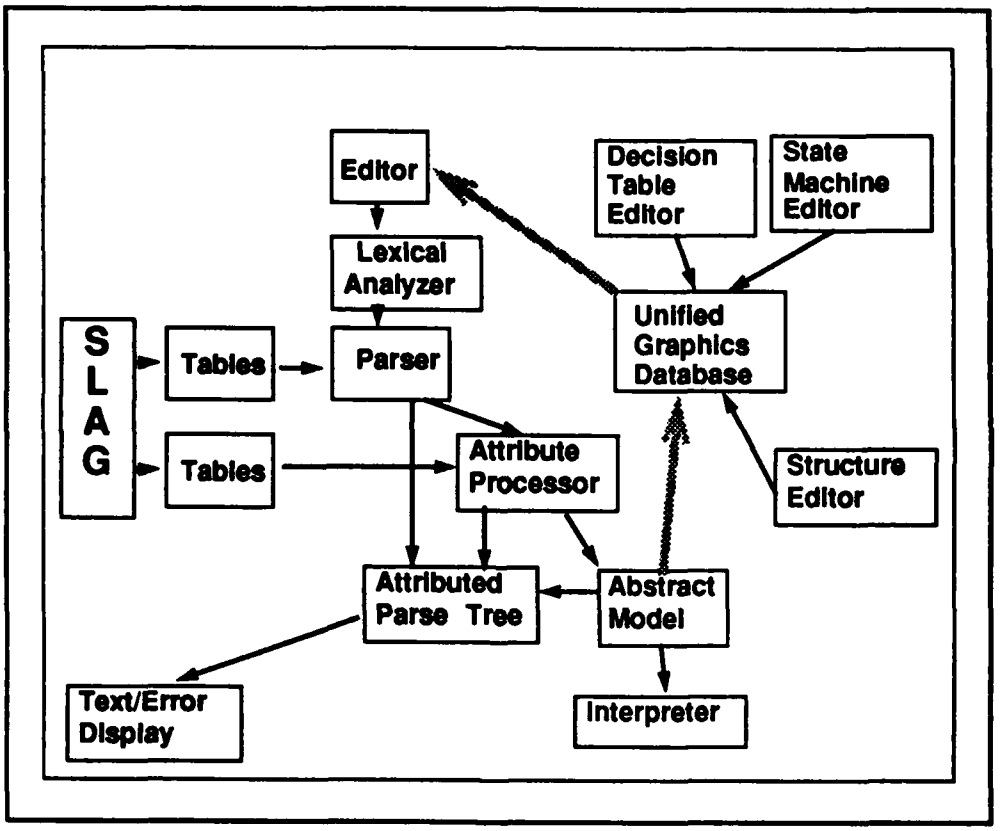


Figure 3-6. P3, P4, L5 Workstation Architecture

3.5.1 Language Processing

An expanded block diagram of the language processing portion of the block diagram is shown in figure 3-7. The language dependent portions of the Parser and IFORM Manager are contained in tables. The content of these tables is determined by an attributed grammar description of the programming language. This description is written in a Scoped Language for Attributed Grammars, or SLAG for short. This formal description is then processed by the SLAG processor to generate these tables.

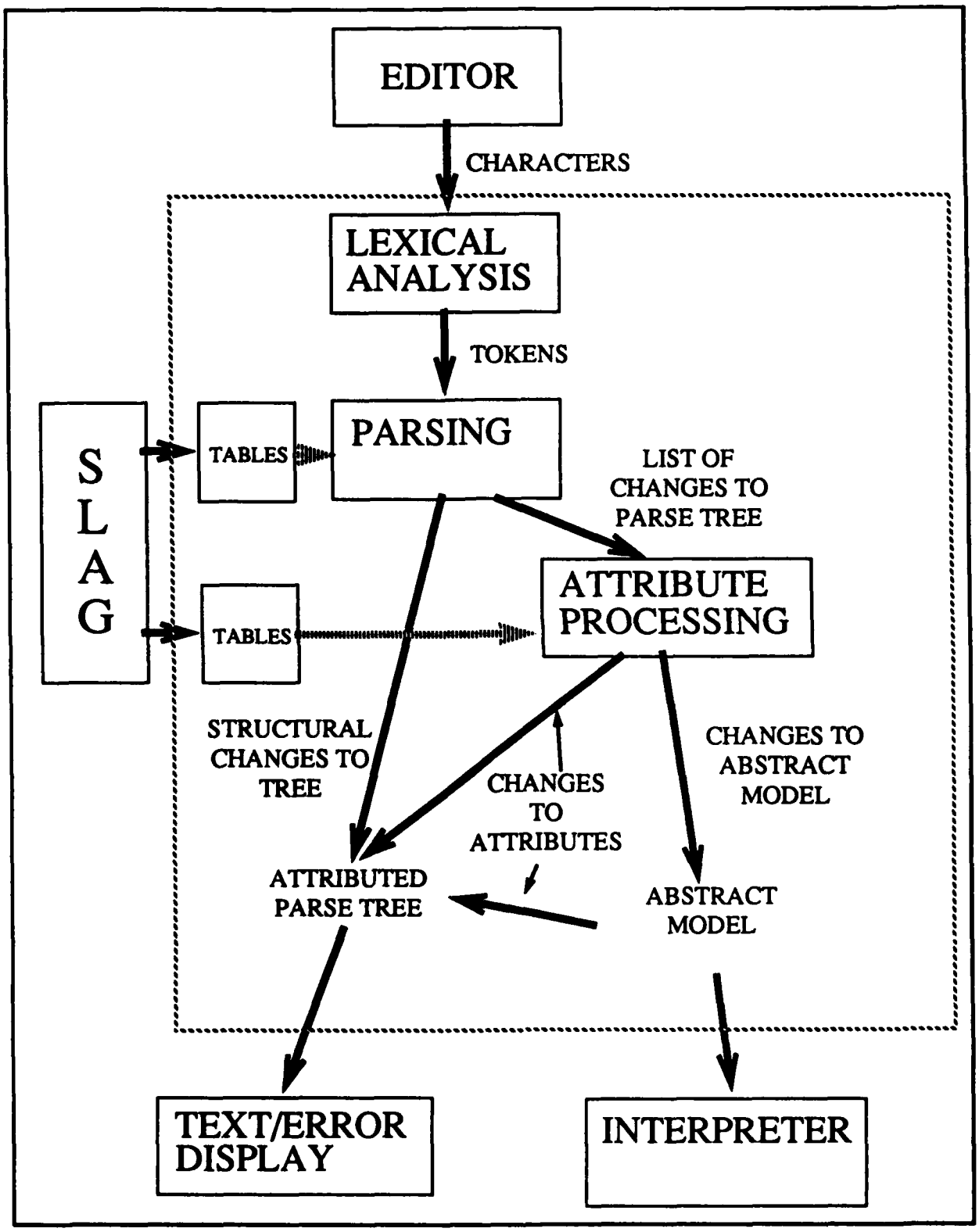


Figure 3-7. P3/P4/L5 Language Processing

3.5.1.1 Semantic Analysis

Semantic analysis actually involves three activities:

- the processing of the lists of changed nodes from the parser;
- the processing of semantic functions; and
- the processing of change notifications in the abstract semantic model.

These activities do not necessarily occur in this order.

Lists of changes that have occurred to the parse tree are provided by the parser. These lists indicate which parse tree nodes have been added or deleted, which nodes have been newly connected into the tree, and which nodes (in the case of terminals) have had their values changed. The semantic analyzer then determines which semantic functions need to be executed (by referencing data from the table) as a result of these changes. Semantic functions whose execution is dependent upon the creation or deletion of parse tree nodes are executed immediately. Other semantic functions are placed in a sorted list. When the processing of the lists of changed nodes has been completed, the semantic analyzer removes and executes semantic functions from the function queue.

The execution of a semantic function usually results in the calculation of a new attribute value for a node in the parse tree. Some of these functions have the side effect of creating or destroying objects in the abstract semantic model. Functions that create objects always return a pointer (handle) to the created object, and this value is placed in an attribute on the parse tree. Some functions are executed for side-effect only: these functions execute methods on objects in the abstract semantic model, and their execution does not result in the direct alteration of any attributes in the parse tree.

Changes to attributes in the parse tree will cause semantic functions in which the attribute appears as an argument to be queued for the IFORM Manager to process.

Objects in the abstract semantic model may monitor other objects. Objects that are changed notify their monitors by placing a notification message on the list of pending change notifications (although not every change to an object warrants a change notification). The IFORM Manager passes pending change notifications to the appropriate abstract semantic model objects. Some monitors in the abstract semantic model cause the updating of attributes in the attributed parse tree.

3.5.1.2 Language Processing Control

The coordination of the activity of the lexical analysis, parsing, attribute processing, and abstract model processing is crucial to the operation of these prototypes. Altering some of the data structures while other operations are in progress can lead to incorrect results. Failure to interrupt the change notification process can lead to unacceptable delays in parsing and text display update (the change notification queue and associated processing were added in the P4 prototype to improve user responsiveness).

Figure 3-8 shows the work queues for language processing control, and the following algorithm presents the logic for coordinating this behavior:

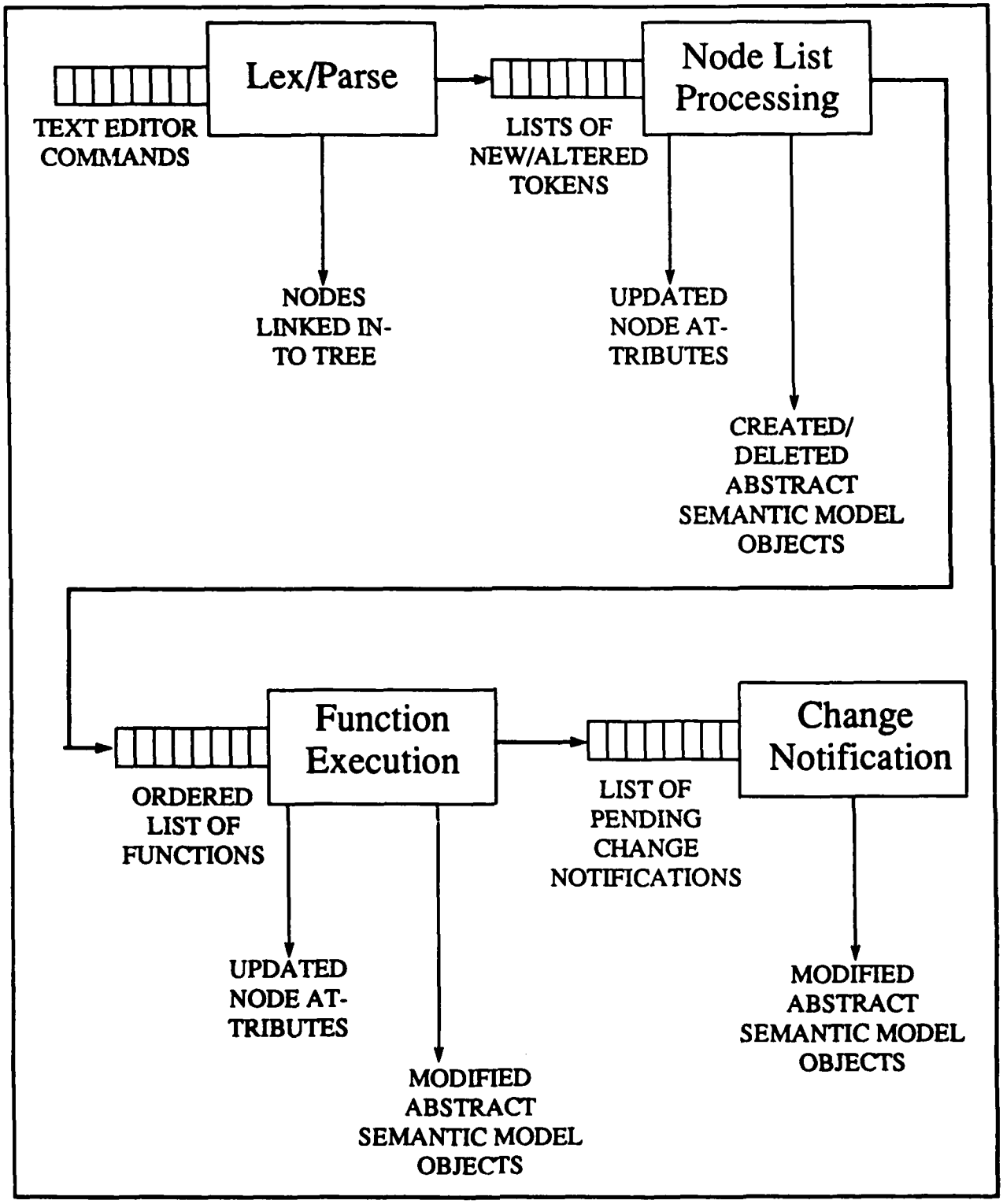


Figure 3-8. P3/P4/L5 Language Processing Control

```
LOOP
    IF messages are present from the editor
        Perform Lex/Parse on all messages
    ELSE IF There are change notifications on queue
        Process an arbitrary (small) number of notifications
    ELSE IF There are new or altered tokens
        Process an arbitrary (small) number of tokens
    ELSE IF There are functions on the function execution queue
        Process a single function
    END IF
END LOOP
```

The design of this loop ensures that the order in which incremental changes are made to the program does not affect the syntactic and semantic analysis of the final resulting program. It further ensures that commands from the editor are processed promptly.

3.6 Incomplete P6 Prototype

Figure 3-9 shows the block diagram of the P6 prototype that was under development at the end of the contract. In this prototype, the structure editor and language processing now share the abstract model instead of having separate databases. The responsibility for recognizing and annunciating semantic errors has shifted from the attributed parse tree and the text display to the abstract model and a separate error annunciation tool.

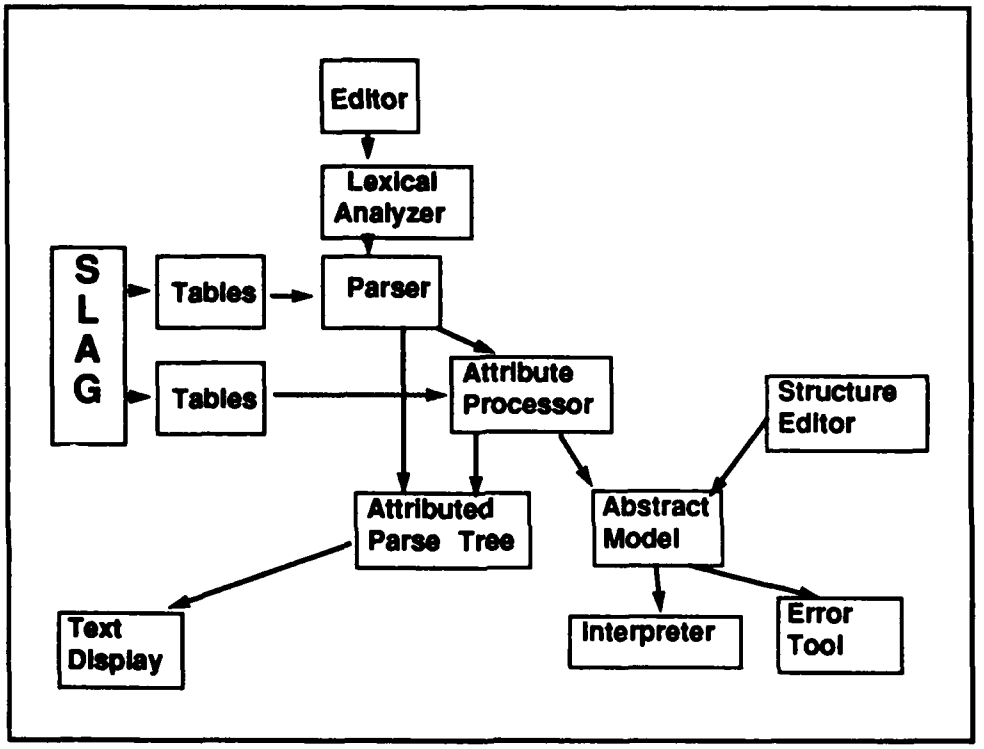


Figure 3-9. P6 Workstation Architecture

3.6.1 Language Processing Control

With the movement of the semantic error annunciation from the attributed parse tree to the abstract model, the control of the parser and attribute processor becomes nearly independent of the abstract model processing. Figure 3-10 shows the revised language control for the parser, and the following algorithm outlines the logic for coordinating this processing:

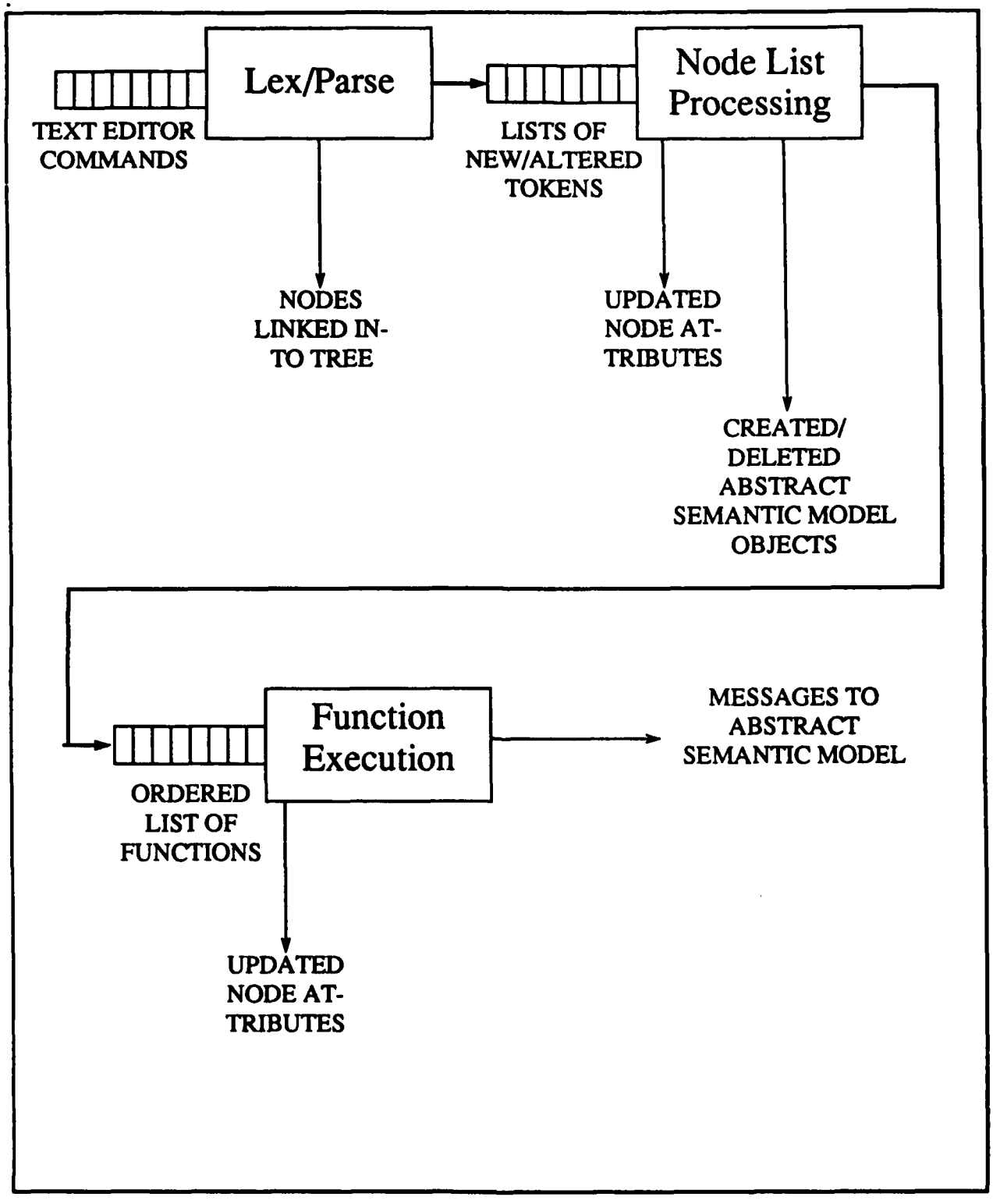


Figure 3-10. P3/P4/L5 Language Processing Control

```
LOOP
    IF messages are present from the editor
        Perform Lex/Parse on all messages
    ELSE IF There are new or altered tokens
        Process an arbitrary (small) number of tokens
    ELSE IF There are functions on the function execution queue
        Process a single function
    END IF
END LOOP
```

Figure 3-11 shows the language control for the abstract model, which is coordinated by the following algorithm:

```
LOOP
    IF messages are present at the input
        Process inputs, creating references and returning handles
    ELSE IF there are transactions to be processed
        Process an arbitrary (small) number of transactions
    ELSE IF There are notifications on the change notification queue
        Process an arbitrary (small) number of change notifications
    ELSE IF There are notifications on the semantic change notification queue
        Process an arbitrary (small) number of semantic error checks
    END IF
END LOOP
```

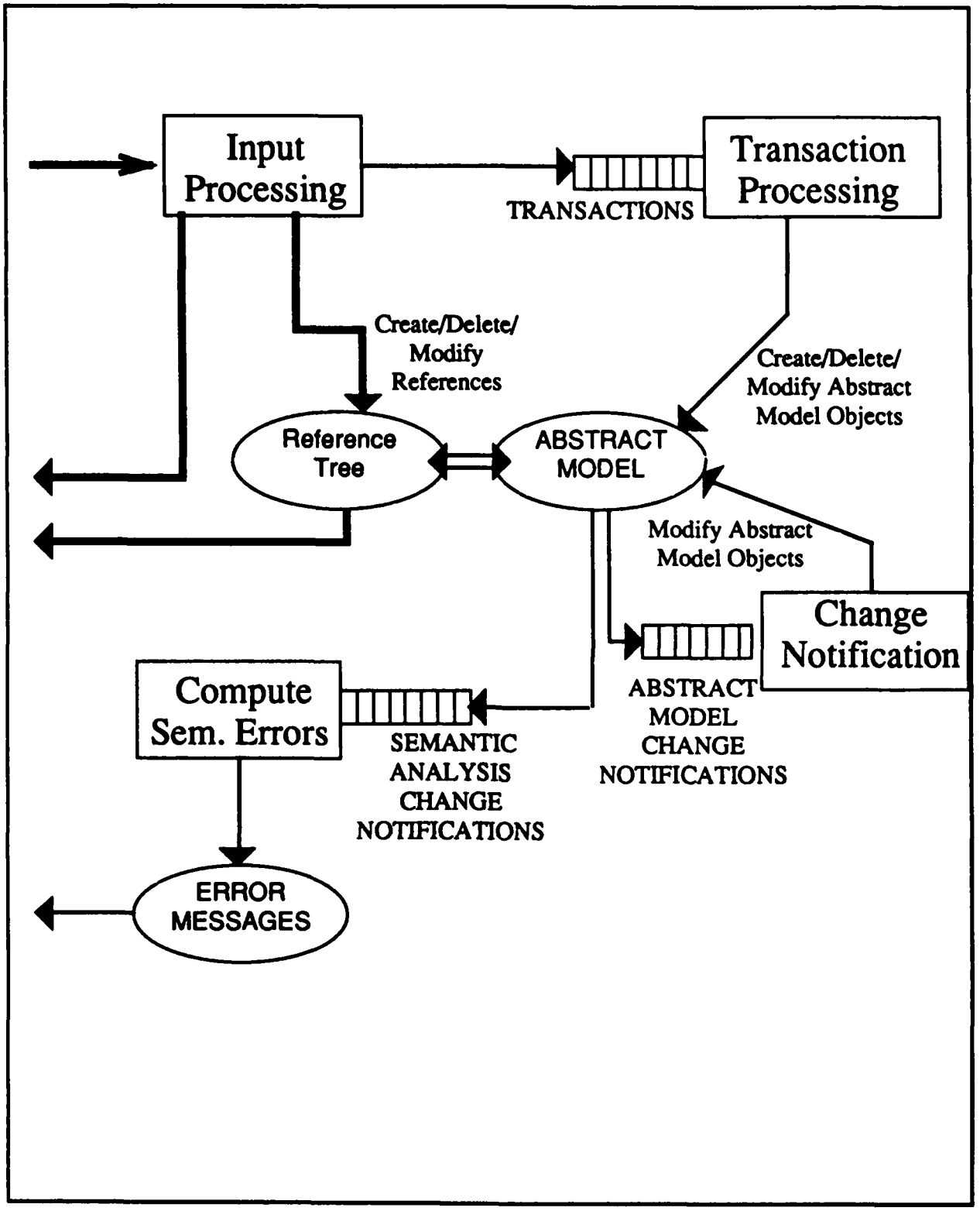


Figure 3-11. Abstract Model Manager

4

Concurrent Update: The Evolution of the Abstract Model and the Synchronization Problem

Summary

This chapter describes concurrent update issues relating to the evolution of the Abstract Model and the synchronization problem.

It includes detailed problem statements and discussion of the architectural model, object selection and the obsolete window map problem discussion and analysis, implementation issues and changes since the initial prototype.

4.1 Problem Statement

The initial structure of the research program embarked upon two separate thrusts (graphics editors and language editors) with an eye toward merging these two thrusts together into a system in which graphic changes were automatically reflected in the language view, and vice versa. The approach currently being undertaken to accomplish this merger is to place all significant design information in a memory-resident database known as the abstract model.

The placement of significant information in the abstract model requires the solution of several design problems. The first and obvious design problem is the definition of an abstract representation that meets the needs of both the language and graphics editors. The second design problem is the coordination of access to the abstract model: if one editor is making changes to the abstract model, when is it safe to allow another editor to have access to the abstract model? The third design problem is the coordination of the deferred processing that occurs within the abstract model (to perform semantic checks) with the external access to the abstract model from the editors.

In this chapter, we will examine the issues associated with coordinating the access of multiple editors to the abstract model. This is called the synchronization problem. The primary requirement here is that database consistency be maintained: that each editor leave the abstract model in a state that can be examined by other editors, and that the state that it is left in meets certain consistency requirements.

The consistency requirements may be met in a number of ways. The choice between these, however, is not arbitrary: different strategies may have distinctly different manifestations from the user's perspective. The challenge in the design is to satisfy the consistency requirement while, at the same time, maintaining a user interface that is understandable, predictable and responsive from the user's perspective.

4.1.1 The Architectural Model

The view that the user sees on the screen, while derived from the information in the abstract model, is, in fact, a copy of that information stored in a form more suitable for display purposes. In fact, there may be two or more of these stored forms representing various stages in the transformation from the abstract model to the screen, and vice versa.

For the purpose of the following discussion, we will simplify the architecture to assume that there is only one transformation occurring between the abstract model and the display that is visible to the user. In this simplified analysis, we view the abstract model as simply an internalized view of textual data stored in buffers. The relationship between the screen display and the buffer is described by a window map that relates visible objects (characters) on the screen to the structured text in the buffer. This simplified view allows us to examine and understand the issues more clearly, and generalizes well to the actual architecture of the IAW.

4.1.1.1 Buffers, Editors and Windows

We begin by stating our assumptions. Our simplified architecture contains:

1. a set of independent buffers containing data
2. a set of editors for invoking commands, and
3. a set of windows for displaying data

We assume that each window is managed by one and only one editor and displays data from one and only one buffer. We further assume that each editor can handle multiple windows, and that different windows associated with the same editor may be associated with different buffers.

From the buffer's viewpoint, each buffer can be associated with (viewed by and altered by) several editors and can be displayed in multiple windows. Buffers contain data about objects and can possess multiple views; for example, a buffer might have a text view and a graphics view. When a change is made to an object contained in one of the physical views, the change should be propagated to all other views.

4.1.1.2 Processing User Commands

Our view of the user interface is that users issue commands and specify objects on which the commands are to perform. Each command is directed to and processed by an editor.

Processing a command can require an update of the window in which the command was issued (e.g., a command to move the cursor within a text window). An editor might request that the responsible buffer manager perform an update on an object. The resulting change in the buffer contents will normally cause an update in the window from which the object was selected. The buffer update may, in turn, cause updates in other windows displaying the objects in the buffer.

4.1.1.3 Window Maps

Each window's editor maintains a window map that describes the mapping of objects from the buffer being displayed in the window and positions in the window. When the mouse is clicked in a window, or a position-specific text-editing command is issued, the window map determines which object is at the specified position.

4.1.2 Object Selection and the Obsolete Window Map Problem

One of the most important aspects in implementing commands is the identification of the selected object for a command. Many commands involve specifying a position within a window, either explicitly or implicitly. If the window map of that window is not up to date, there is some danger that a user can select an item in a window display based on incomplete or obsolete data.

In a window based system, the user can typically select an object in the following ways:

1. mouse selection
2. cursor position
3. position relative to cursor

There are several ways in which the window map may be out of date. These are explored in the following sections.

4.1.2.1 Object Selected by Editor No Longer Exists

When a user selects an object by specifying a position in a window, the editor refers to a window map to determine which object is at the specified position. There is a danger that the selected object no longer exists because a previously issued command caused it to be deleted from the buffer.

4.1.2.2 Viewed Object Not Selected by Editor

There is also a danger that a user could select an object on the basis of the current display and the editor selects a different object due to a change in the buffer and/or window map between the time the user makes the selection and the time the editor converts the user input into the identity of the selected object.

4.1.2.3 Disabling User Input

An editor might sometimes avoid object selection problems by disabling user input devices. When user input devices are disabled, user input relative to object selection is ignored and user input in the type-ahead stream is discarded. However, disabling user input could lead to a clumsy user interface.

4.1.2.4 Object Selection Policies

The issue of what should happen in the case of a user trying to select an object while the buffer and/or window map might be changing can be resolved in a number of ways:

1. Conservative Policy - allow user input only when there is no possibility of incorrect object selection

2. Buffer-Oriented Policy - base each object selection on the buffer that results from performing the previous command

3. Display-Oriented Policy - base each object selection on the display seen by the user at the time the selection is made

4. Opportunistic Policy - base each object selection on the window map that exists at the time the editor interprets the command.

Object Selection Policy Properties

Policy	Selected Object Always Exists	Viewed Object Always Exists	Input Always Enabled
Conservative	Yes	Yes	No
Buffer-Oriented	Yes	No	Yes
Display-Oriented	No	Yes	Yes
Opportunistic	No	No	Yes

Figure 4-1. Object Selection Policy Properties

4.1.2.5 Recovering From the Selection of a Nonexistent Object

For both the display-oriented policy and the opportunistic policy, it is possible that the user's selection of an object will be interpreted by the editor as an object that no longer exists. The editor must be designed to accommodate this issue.

4.1.2.6 Handling Type-Ahead

Type-ahead is the rapid issuance of a series of commands at a pace faster than the display can accommodate. It can involve both keyboard and mouse input. The object-selection policy determines the manner in which type-ahead is handled.

4.2 Problem Discussion and Analysis

Each of the possible object selection policies outlined in the previous section has desirable and undesirable characteristics. With the conservative policy, user input is disabled whenever there is any possibility of the screen being inconsistent with the internal representation. While this approach guarantees that the operations are always well-defined, it has the annoying side-effect of disabling and reenabling the user input based upon the state of the processing. The behavior of the system will thus change based upon the rate at which user input occurs relative to the internal processing speed. From a user's perspective, this is perceived as inconsistent behavior (it should not matter how fast you type!).

The buffer-oriented policy escalates objects at the time that the editor executes the command, and after the screen has been updated with the results of the previous command. This policy has the disadvantage of possibly operating on a different object than the user selected, if the execution of another operation changes the screen contents. However, this is exactly the way type-ahead operates, and users appear comfortable with how type-ahead works. This policy has the advantage that the selected object always exists.

The display-oriented policy selects the object to be operated on at the time that the user enters the command. While there is no ambiguity in the user's intention, this policy has the disadvantage that the selected object may not exist when it is time to execute the command if an intervening command has deleted the object. Now the system designer must define the behavior of the system in this case, and make it obvious to the user at the same time. No good understandable solution that implements this policy was discovered.

The opportunistic policy is the same as the buffer oriented policy, save that the screen is not necessarily updated with the results of the previous command. This policy has the disadvantage that it can select an object different from the one the user originally indicated, and can also result in the section of an object that no longer exists. It thus has all of the disadvantages of both the display oriented and buffer oriented policies, and results in an interface that is very confusing to the end user.

Based upon the above reasoning, the buffer oriented policy was selected as the policy to be used with all editors.

4.3 Implementation

The following description of the concurrent update implementation is based upon several abstract concepts: a notion of an active editor; a notion of an entity manager, which encompasses the scheduling aspects of the operating system; and the notion of a transaction manager, which handles communications between the various processes.

An editor will seek to become active when user input is directed to a window that it is managing. The editor will request active status (which must be yielded by the editor that currently has this status) from the transaction manager. When all processing associated with the current active editor has been completed, and the transaction manager has sent all pending update messages to the editor requesting active status, it will grant active status to that editor. Once the editor has processed these update messages, it is guaranteed that its window is consistent with the internal representation.

Once the editor is designated as the active editor, it has exclusive read/write access to the internal representation until it yields such permission. In general, the editor will not yield permission when it has pending user input.

5

Abstract Semantic Model Structure

Summary

This chapter explains the structure of the Abstract Semantic Model. It begins with a description of how the Abstract Model evolved, and includes data on the impact of concurrent update. It contains a section that provides an overview of the Abstract Model including descriptions of:

- Elements, sets and monitors
- Derived sets
- Masking union, and
- Indices

Environments and references are described in detail including: additional visibility checks, types (mathematical models of types and abstract data types).

5.1 Evolution of the Abstract Semantic Model

The term IFORM was originally used in the P1 language prototype (delivered with the P2 prototype) to refer to the *internal form* or representation of the program that was being used by the language processing tools. As the prototypes advanced, this notion of an internal form split into two distinct notions: an attributed parse tree associated with the parser, and an abstract representation of the program that became known as the abstract semantic model, or simply the abstract model.

While it is possible to describe the entire compile-time semantics of a language with an attributed grammar, some significant computational problems are associated with the implementation of this approach.

First, if a piece of data is required some distance away in the tree, it must be replicated many times on intermediate nodes in the process of moving it from the source node to the target node. The sheer volume of data that must be moved makes this approach impractical.

Second, most of the information that needs to be moved consists of relationships between data elements as well as the data elements themselves. For example, when a function is declared, its identifier and return type, as well as an ordered list of the identifiers, types and modes of its formal parameters, needs to be communicated to all potential reference points in the program.

The abstract semantic model was conceived as a solution to both of these problems. The abstract semantic model gathers related data elements and relations together into objects known as *Descriptors*. Instead of passing these descriptors from node to node, they are placed in a set (whose identifier is passed from node to node) representing the "context" in which the identifier is defined and referenced. Daemons (active functions) known as Monitors then monitor these sets to perform semantic analysis.

In the P3, P4 and L5 prototypes, the semantic analysis performed by the abstract model was limited to visibility computation and overload resolution. The remaining semantic checks were performed by the attribute processor associated with the attributed parse tree. This resulted in both a complicated interface between the parser/attribute processor and the abstract model, and a very complicated control structure that led to annoying delays from the user's perspective. In the prototype that was under development when the stop work order was received, all semantic checking was being migrated into the abstract model. This allows the parser to give immediate syntax feedback to the user without waiting for the semantic analysis to reach a stopping point. Furthermore, it led to a significant simplification in the interface between the abstract model and the parser/attribute processor.

5.1.1 Impact of Concurrent Update

As the design of the language processing elements neared completion, and the issue of concurrent update was considered, the abstract model began to take on a different role: that of intermediary between the language and graphics views. Chapter 6 addresses the extensions to the abstract model necessary to support the graphics view.

5.2 Overview of the Abstract Model

There are really two layers to the abstract model: the level at which it presents a logical model of a program; and the level of primitives that provide the mechanisms for incremental semantic analysis.

At the logical level, a program is viewed as a collection of declarations and references to those declarations. The semantic analysis problem is essentially to determine which declaration each reference refers to, and to further ensure that the declaration is the type of declaration required.

Each declaration in a program is modeled in the form of a descriptor in the abstract semantic model. Each instance of a package, procedure or function declaration has a corresponding descriptor representing that particular instance. Similarly, instances of type, literal and variable declarations are also modeled as descriptors. These descriptors contain information about the declaration, such as what kind of declaration it is, naming information, and a pointer back to the parse tree node that created it.

References, on the other hand, represent a need for semantic analysis. In the abstract model, they are represented by a collection of objects that actually perform the semantic analysis. These collections are called references. In the following subsection, we will examine the nature of the objects that perform the semantic analysis.

5.2.1 Elements, Sets and Monitors - An Overview

Conceptually, semantic analysis can be viewed as a data flow diagram, with each node in the data flow diagram representing a computation that must be performed, and the arcs between the nodes representing the data that passes between the computations. In our abstract model, we represent the data flowing between the nodes as sets, and the computations being performed by monitors. Monitors that produce results for use by other computations are modeled by a combination of a monitor and a set, where the set represents the result of the computation. These combinations of monitors and sets are known as derived sets.

The basic building blocks of the abstract semantic model are elements, sets and monitors, where an element is any object that is a member of a set. A monitor is a daemon (active function) that is attached to a set or element, and monitors the set for changes. If the monitor detects a change, it executes specific functions. The function may revise the contents of another set, or produce a boolean result indicating a semantic error.

☞ *Note that while elements of sets are usually descriptors representing program declarations, sets may in fact contain many other kinds of objects, including other sets and raw data elements.*

The function associated with a monitor is triggered when the set to which it is attached changes. The specific action that the monitor takes can be dependent upon the nature of the change: was an element added or removed from the set, or did an element already in the set simply change in some unspecified way. Note that in the event of an element changing, the monitor is not informed of the nature of the change, only that the set has changed.

If we let a set contain representatives of all declarations that have occurred, we could perform semantic analysis by attaching monitors to the set. For example, consider the following program:

```
PACKAGE BODY example1 IS
    TYPE X IS (first, last);
    A : X;
BEGIN
    A := first;
END example;
```

The following figure shows the set for this example containing four descriptors:

1. type X,
2. variable A,
3. literal first
4. literal last.

The figure also shows an example of a semantic question that would arise from the assignment statement: once the left-hand side of the assignment statement has resolved to a variable with a particular type,

does the literal on the right resolve to a single descriptor?

This question is embodied in a monitor that sets an error flag if the answer is no.

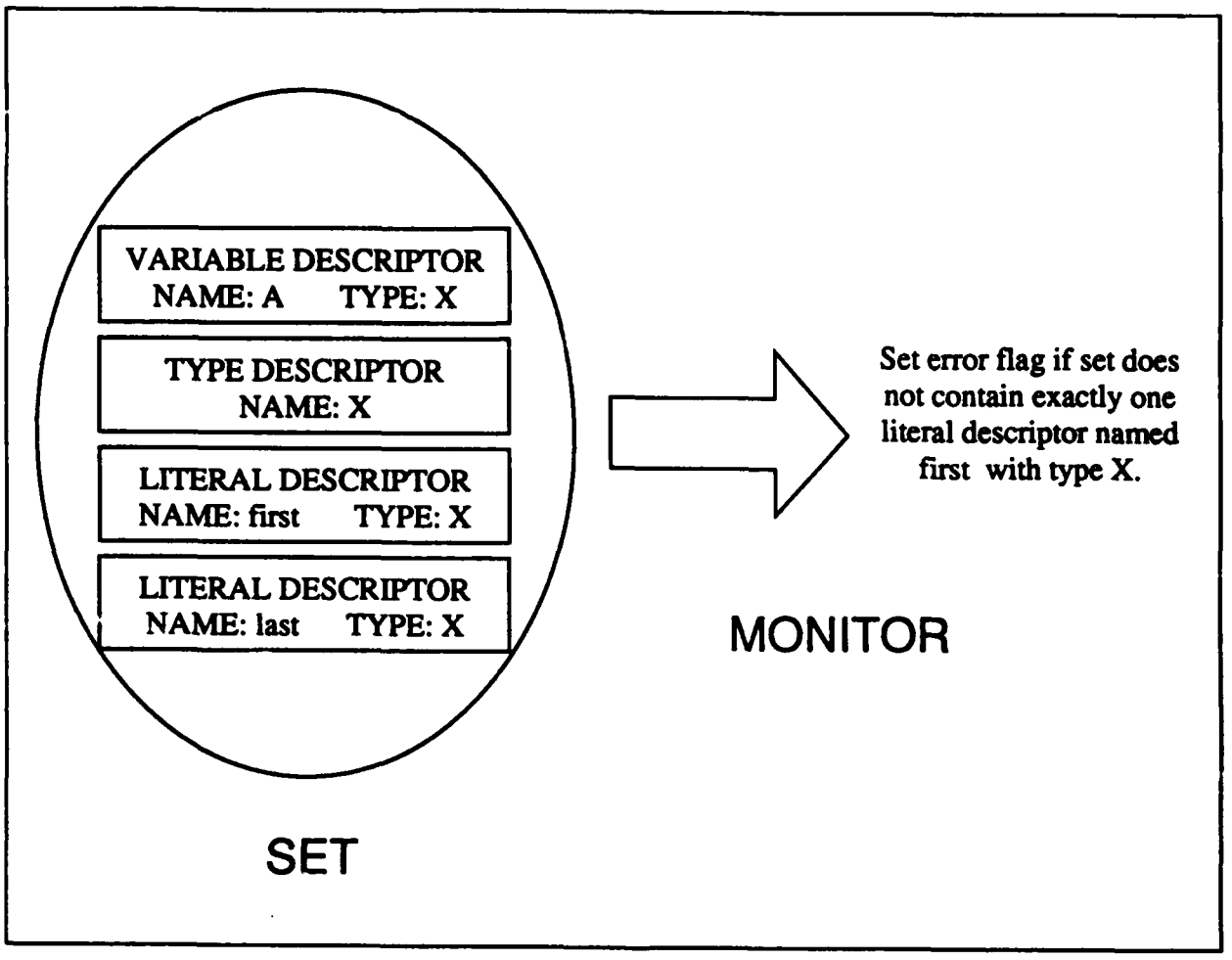


Figure 5-1. Abstract Semantic Model, Example 1

5.2.2 Derived Sets

This approach to semantic analysis suffers some computational problems. Each time the semantic question must be reevaluated (which is each time any change occurs to the set), the monitor must:

- Determine which descriptors are visible
- Determine which of these descriptors have the desired name (with complexity proportional to the number of visible descriptors)
- Determine which of the descriptors with correct visibility and name have the correct type signatures.

If an entire program were modeled in this manner, the computational complexity would approach $O(n \cdot r)$, where n is the number of descriptors, and r is the number of references (monitors).

☞ *The amount of code that needs to be executed in each case also implies that this relationship has a fairly large constant associated with it.*

To begin to reduce this complexity, we introduce a new notion called a derived set.

A derived set consists of a set and a monitor that determines its contents. For example, we could create a derived set containing only those descriptors from the direct environment whose name is "first."

☞ *The determination of membership of a given element in the derived set is independent of the size of the source set, and is a relatively simple function. With this approach, we can reduce the size of the constant associated with the complexity of the analysis.*

While virtually any function can be used to determine membership in a derived set, in practice most derived sets fall into one of three basic categories:

1. boolean sets,
2. filtered sets, and
3. mapped sets.

A boolean set takes two or more source sets and performs a boolean function (the monitor) on their elements: intersection, union, or difference.

In a filtered set, the monitor is a predicate function applied to each element in turn. Membership is based upon the predicate's boolean result. This can be used to select descriptors based on name, return type, or other properties.

The monitor for a mapped set takes a descriptor in the source set and maps it into a value in the derived set. The most common map takes a descriptor into its return type, so that the derived set is now the set of return types represented by the source set.

While these basic types of derived sets serve most needs, they are frequently combined to form more complex functions. For example, a filter and a map could be combined to conditionally map descriptors into values.

The determination of the contents of a derived set is done in two steps.

1. When the set is initially created, the descriptors that satisfy the membership conditions are placed in the set.
2. After this initial load of the set, the derived set is informed of any subsequent additions or deletions to the source set, or of changes that occur to any existing members of the source set. When these messages are received, the monitor associated with the derived set evaluates the changes to determine if the contents of the derived set should change.

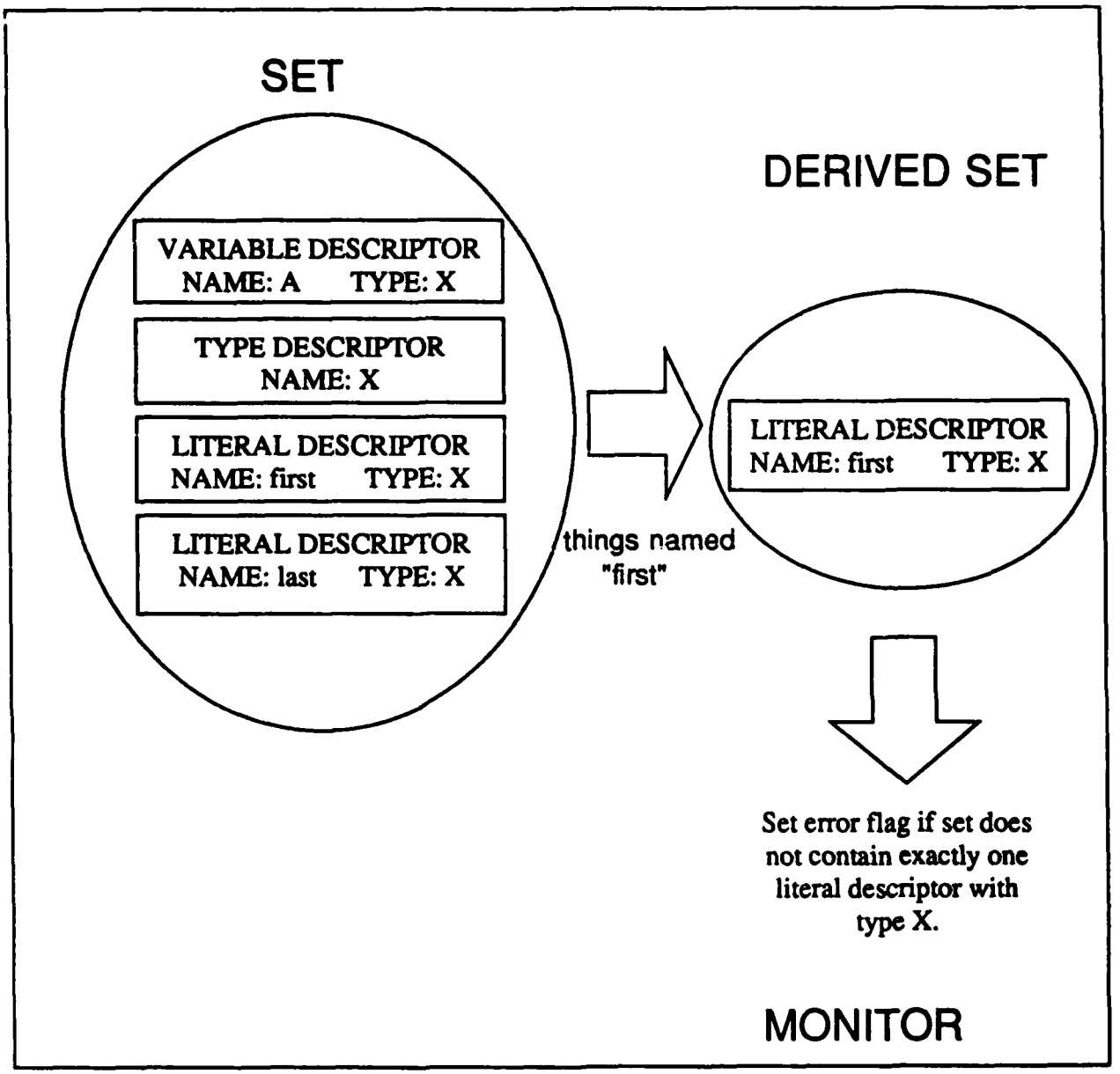


Figure 5-2. Abstract Semantic Model, Example 2

5.2.3 Masking Union

While implementing the Ada rules for determining the visibility of a declaration in a program, it became apparent that a derived set was required in which elements from one source set would take precedence over (dominate) elements from a second set. This set became known as a masking union, because elements from the dominant set can mask the existence of elements from the recessive set. The set operates as follows:

Given a predicate function (known as a homograph test in Ada) that determines if two elements are "indistinguishable" according to some criteria, the following rules determine membership in a masking union:

A member of the dominant set is a member of the masking union if and only if it has no homograph in the dominant set. In other words, if two elements in the dominant set are indistinguishable, neither becomes a member of the masking union.

A member of the recessive set is a member of the masking union if and only if it has no homograph in either the dominant or recessive set.

The definition of homograph used in the Ada language is as follows:

DEFINITION: Homograph - Two declarations are homographs of each other if either:

a) they have the same identifier and overloading is allowed for at most one of the two

b) they have the same identifier and parameter and result type profile

Figure 5-3 presents an example of a masking union using the Ada homograph definition.

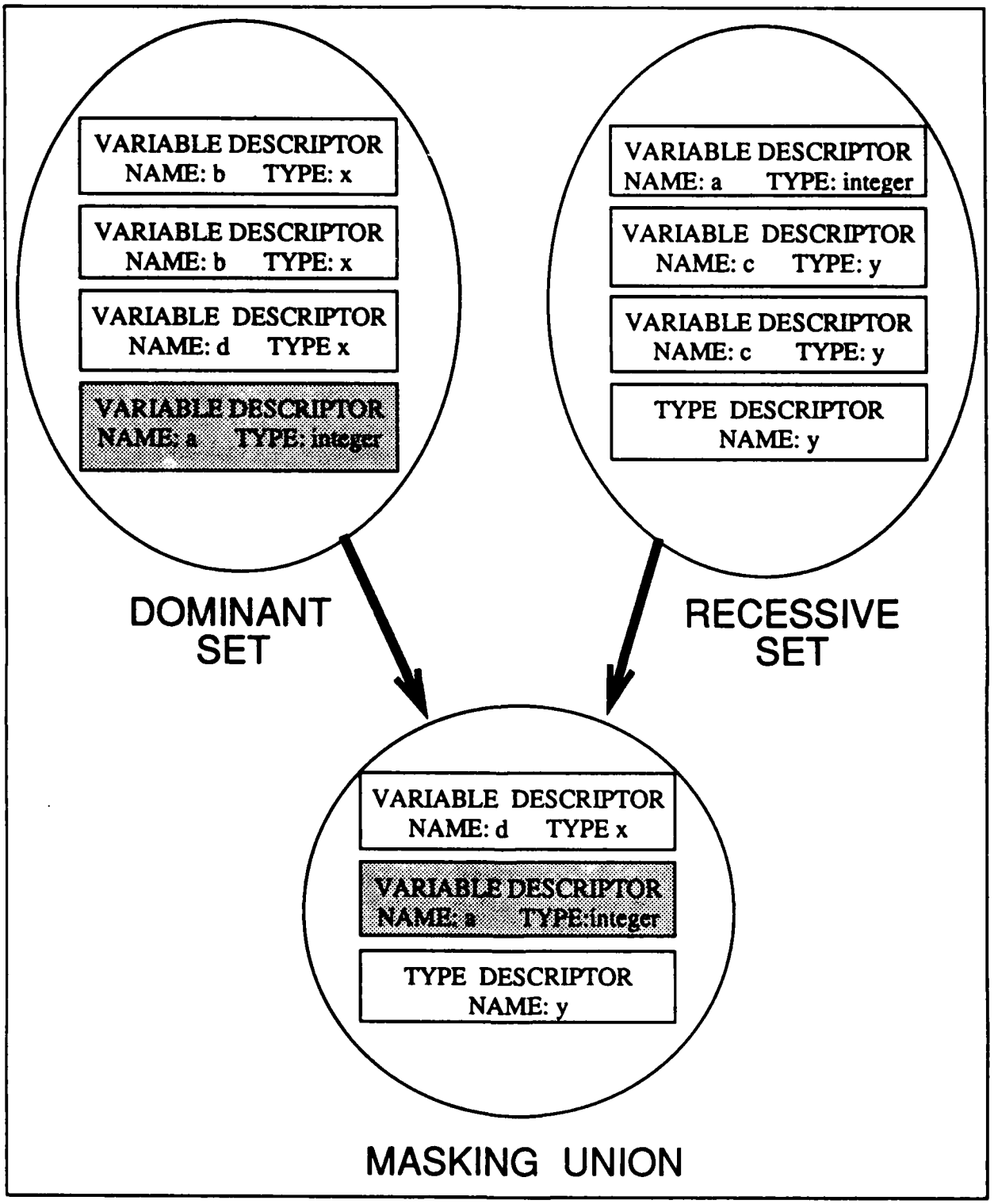


Figure 5-3. Masking Union with Ada Homograph Definition

5.2.4 Indices

During the implementation of the abstract semantic model, it became apparent that the determination of membership in masking unions could be computationally improved by adding indices to the dominant and recessive sets. Accordingly, a generalized indexing capability was added to all sets. In general, indices on a particular key are created the first time an indexed access is attempted with that key. In this way, indices are not created until they are needed. In addition, to facilitate the processing of changes to individual elements, an inverse index is kept so that, should a key value change, the old index key may be recovered. This was necessary for masking unions so that descriptors that might have been "hidden" can now become visible.

5.3 Environments

The concept of an environment is frequently used in describing the semantics of a programming language. In the formal sense, an environment is a mapping of identifiers to the declarations that they designate. When an identifier is encountered in an expression, the environment is then used to determine which declaration(s) could possibly be designated by the identifier (overload resolution within the expression may be required to narrow down to a single declaration). In conventional compilers and assemblers, the environment is often referred to as the symbol table.

One of the more difficult computations in the incremental semantic analysis of a scoped programming language is the determination of the environment at a particular point in a program. Batch compilers typically both build and use the symbol table as they progress through a program, adding declarations as they are encountered, and removing them as the scope changes. Since the table is being used at the same time, when an identifier is encountered, the symbol table represents the actually visible declarations at that point in the program.

Since identifiers (other than the name of the thing being declared) can occur in a declaration, the environment used to interpret the identifiers in declarations is different for each declaration. For example, if there are d declarations, there are d+1 significant environments, each differing from the previous one by a single declaration. This does not present a problem to the batch compiler, for it successively modifies and uses the table as it passes through the program. However, it presents a serious computational problem for the incremental compiler.

The problem is that in order to evaluate an incremental change, the symbol table must be reconstructed at the point of change. If no symbol table information is saved, the entire program must be reanalyzed from the beginning in order to reconstruct the symbol table at the point of change. If all symbol table information is saved, then no time is required, but $O(d^2)$ space is required, where d is the number of declarations in the program. If key symbol tables are saved, then the table at the point of change must be reconstructed from the nearest checkpointed symbol table (space may still be $O(d^2)$). In any case, if the change is simply a new occurrence of an identifier, then it is analyzed and the change is complete. If, however, the change is a new declaration, then the remainder of the program that is within the scope of the new declaration must be searched for identifiers whose interpretation may be affected by the new declaration. Furthermore, if there are saved environments, those that are affected by the change must be resaved. The net effect is that the worst case incremental editing scenario (without considering the complexity of overload resolution) has $O(d+r)^2$ complexity, where d is the number of declarations, and r is the number of references.

One of the major sources of complexity here is the large number of environments that occur in the declarative regions of a program due to the interleaving of references and declarations. The IAW takes a slightly different approach to environments.

This direct environment is actually computed from the declarations that occur locally within the declarative region (local declarations) and the direct environment that is inherited from the next outer scope.

A masking union (see figure 5-4) can be used to describe this computation with:

- the dominant set being the set of local declarations,
- the recessive set being the direct environment from the next outer scope, and
- the masking union itself being the current direct environment.

Visibility computations for Ada are further complicated by the presence of both specifications and bodies, each of which has its own declarative region, and the Ada "use" clause. Figure 5-5 presents the full environmental computation in terms of abstract semantic model sets.

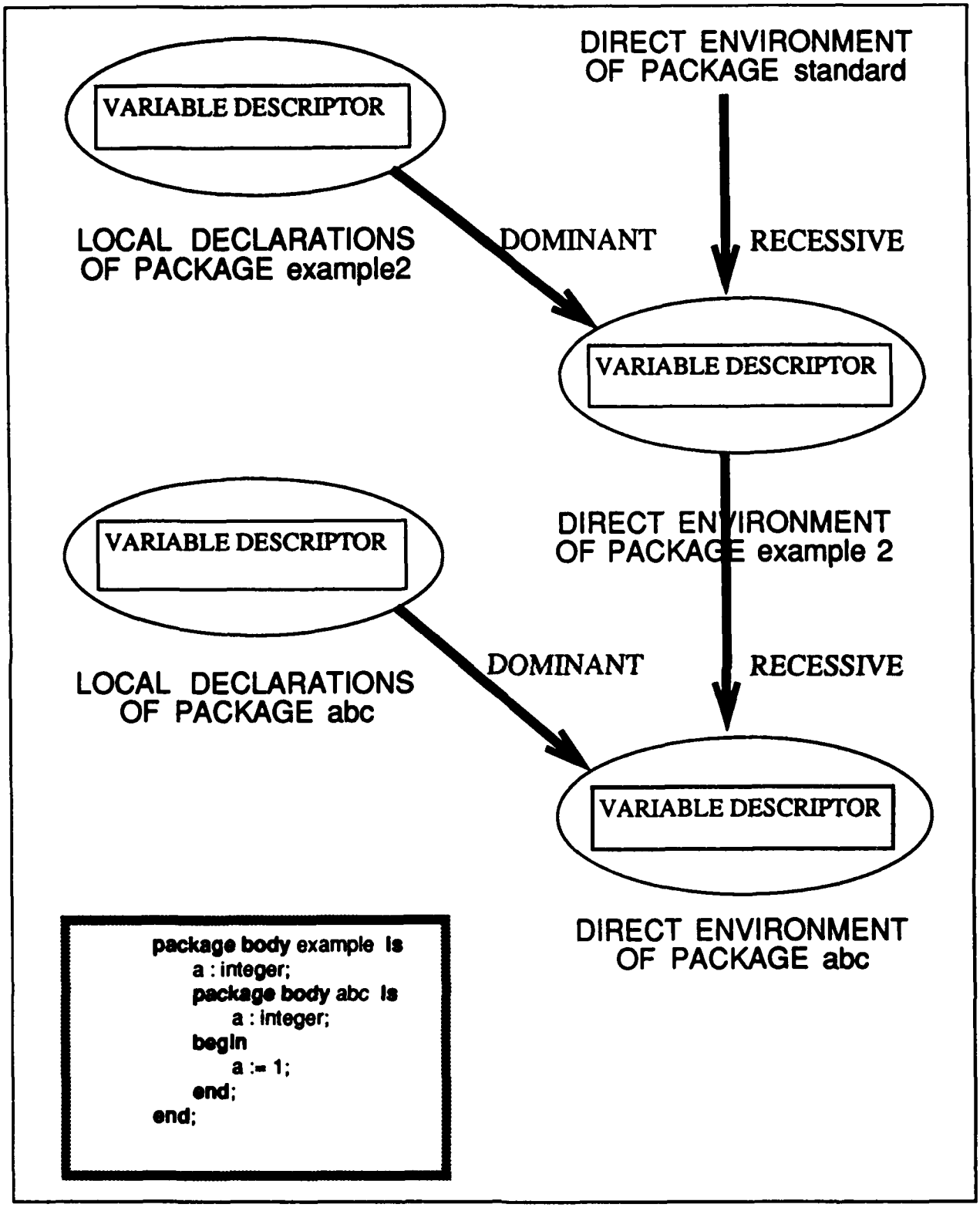


Figure 5-4. Direct Environment Computation with Masking Unions

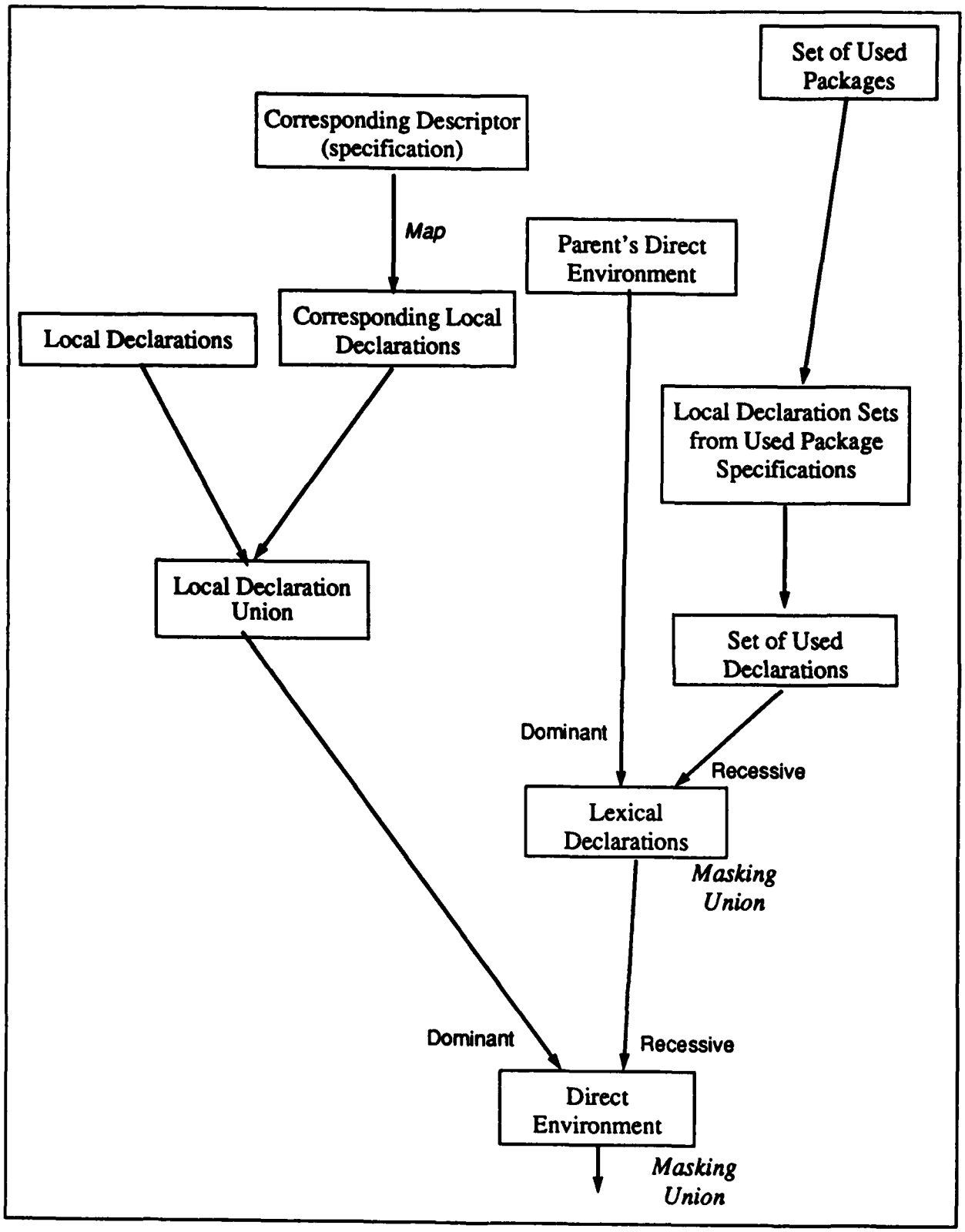


Figure 5-5. Full Ada Body Direct Environment Computation

5.3.1 Additional Visibility Checks

The following visibility rules are not enforced by the direct environment computation, and must be checked by other means:

> "Within the specification of a subprogram, every declaration of the same identifier as the subprogram is hidden; the same holds within a generic instantiation that declares a subprogram, and within an entry declaration or the formal part of an accept statement; where hidden in this manner, a declaration is visible neither by selection nor directly." (LRM Section 8.3, paragraph 16) This requirement is met by creating a special environment for each declaration.

This environment consists of a masking union whose dominant set contains a "dummy" descriptor whose name is the name of the object being declared, and whose recessive set is the normal direct environment. In this manner, any inappropriate references will resolve to the "dummy" descriptor, and can be readily detected as errors.

> Homographs in the local declarations are allowed under the following circumstances (LRM Section 8.3, paragraph 17):
>
> "Exactly one of the two elements is the implicit declaration of a predefined operation. Note that implicit declarations are not entered as part of the program, but are implicitly declared when a new or derived type is declared (LRM Sections 3.3.3 and 4.5). In this case, the implicit declaration is always hidden by its homograph."
>
> and/or
>
> "Exactly one of the two elements is the implicit declaration of a derived subprogram. A derivable subprogram (LRM Section 3.4, paragraph 11) is a user-defined subprogram that redefines an operation that is implicitly defined for a type. A new type that is derived from this type also has this user-supplied operation implicitly defined, and this operation is known as a derived subprogram. In this case, the derived subprogram hides the implicit operator of the same name, and, in addition, is itself hidden by any explicit declaration made by the programmer."

5.3.2 Types

There are several notions of type and of type relationships, all of which are significant to some degree in the IAW. These include the notions of a mathematical type, the abstract data types, derived types, type inheritance and subtypes.

The following sections examine these concepts, and their relationships to the IAW model of types.

5.3.2.1 Mathematical Models of Types

The mathematical notion of type is that of a domain of values. For example, the domain of integers is a type, and the domain of reals is another type. When types appear explicitly in programming languages, what actually appears in the language is a representative of the domain. For example, we might use INTEGER to denote the set of integers, and REAL to denote the set of real numbers. In the mathematical model, all type representatives of a given domain are equivalent: if INT and INTEGER both represent the domain of integers, and the variable a is declared to be of type INT, and the variable b is declared to be of type INTEGER, a may be assigned the value of b, and vice versa.

In the mathematical model, functions and procedures also have types. The type of these constructs is defined in terms of the types of their parameter types and return type, and is referred to as a type signature. For example, the addition operator for integers would have a type signature:

$$\text{INTEGER} \times \text{INTEGER} \Rightarrow \text{INTEGER}$$

Since type signatures are written in terms of domain representatives, we must further stipulate, in the mathematical model, that signatures constructed from equivalent domain representatives are themselves equivalent. Thus the following type signatures are all equivalent:

$$\text{INTEGER} \times \text{INTEGER} \Rightarrow \text{INTEGER}$$

$$\text{INT} \times \text{INT} \Rightarrow \text{INT}$$

$$\text{INTEGER} \times \text{INT} \Rightarrow \text{INTEGER}$$

$$\text{INTEGER} \times \text{INTEGER} \Rightarrow \text{INT}$$

The signature approach can also be used to describe complex data types. A record, consisting of an integer and a real number, would have type:

$$\text{INTEGER} \times \text{REAL}$$

An array of four integers would have type:

$$\text{INTEGER} \times \text{INTEGER} \times \text{INTEGER} \times \text{INTEGER}$$

5.3.2.2 Abstract Data Types

A different notion of a type deals with the expectation that the set of values (the mathematical model) collectively share some properties or operations. For example, for the set of integers, we expect to have a successor function that, given an integer, returns the next integer in the sequence. This notion of a type is then a combination of a set of values (the mathematical model) and a set of operations, and is usually referred to as an abstract data type.

While we may intuitively expect that the operations associated with an abstract data type take values of that type as arguments, it is not mandatory that they do so. For example, an operator for a finite enumeration type may take the type representative itself as an argument, and return the cardinality of the enumeration. As the data type itself gets more complex (a record, or array, or a more complex data structure), intuition begins to fail in terms of what the "type" of the operators should be. As a result, there are, in general, no restrictions placed upon the "types" of operators associated with an abstract data type. In fact, in the extreme, an abstract data type may not represent any data type at all, but may simply be a collection of operators.

The notion of equivalence in abstract data types is much more complex than in the mathematical model, to the extent that the notion is unusable in present programming languages. The reason is as follows: to extend the notion of equivalence completely, we would have to say that two abstract data types are equivalent if and only if:

a) they represent the same value domain

b) they have an equivalent set of operations, by which we mean that each pair of operations (one from each abstract data type):

1) shares the same name

2) shares the same type signature (up to equivalence)

3) have mathematically equivalent functions, which is to say that, for all inputs, they produce the same result and the same side-effects.

Unfortunately, it is an open question as to how to determine the mathematical equivalence of two pieces of code, and therefore, in general, it is not computationally feasible to establish condition b3 above. Consequently, the notion of equivalence is not used in languages that employ abstract data types.

Without the notion of equivalence, type signatures for functions and complex data structures take on a slightly different character. As in the mathematical mode, type signatures are constructed from the representatives of the abstract data types. However, since it is computationally infeasible to recognize type equivalence, signatures constructed from equivalent abstract data types are not equivalent. Thus we have the following results: using the same example as in the mathematical model of types, if INT and INTEGER are two abstract data types representing the set of integers, and the variable, a, is declared to be of type INT, and the variable, b, is declared to be of type INTEGER, a may not be assigned the value of b, and vice versa. In addition, the following type signatures are not equivalent:

$INTEGER \times INTEGER \Rightarrow INTEGER$

$INT \times INT \Rightarrow INT$

$INTEGER \times INT \Rightarrow INTEGER$

$INTEGER \times INTEGER \Rightarrow INT$

5.3.3 Type Environments - An Introduction

We now turn to an abstraction that we will use initially to model abstract data types, and later extend to model other programming language constructs as well, such as subprograms and Ada packages. In our earlier discussion of direct environment computations, we had a notion similar to that of the set of operators associated with an abstract data type, namely the set of *local declarations* in a declarative region. Thus an abstract data type could be modeled as a logical declarative region that contains the operator declarations. Furthermore, if there is an object that serves as a representative of the environment, this object can serve as the representative of the abstract data type. In the IAW these objects are called *type environments*.

Descriptors that represent the operations (functions and procedures) associated with the abstract data type are placed in the *local declaration set* of the type environment. The handle of the type environment object itself serves as the representative of the abstract data type in type signatures.

Type environments consist of two components – specification and body. The specification of the abstract data type contains the specifications of the operations associated with the type. The body contains the bodies (implementation details) of these operations. An example (in pseudo-code) of a "stripped-down" integer abstract data type might be:

```
abstract-type abs-integer is
    function next (a:integer) return integer;
end abs-integer;

abstract-type body abs-integer is
    function next (a:integer) return integer is
        return a+1;
    end next;
end abs-integer;
```

In implementation, a type environment is a complete direct environment, as described in section 4.1. The direct environment of the abstract data type represents the lexical environment in which the operations contained within the abstract data type are declared.

5.3.4 Type Inheritance

An extension to the type environment model allows for type inheritance, enabling the definition of new types in terms of existing types. To implement this, we extend the concept of local declarations to include declarations that are inherited from other abstract data types, with the provision that actual local declarations can override their homographs in the inherited declarations. A composite declarations set (see figure 5-6) is added to the environment computation. This set is a masking union in which the local declaration union is the dominant set, and the inherited declarations form the recessive set.

In this model, the specifications of type environments inherit declarations from the specifications of the inherited type environments, and bodies of type environments inherit declarations from the bodies of the inherited type environments. Local declarations will

mask out homographs from their parents, thus providing local override capability. Note that all declarations, whether from specifications or bodies, end up in the body's local declaration union, and thus are available for use in the body.

Inheritance introduces the notion of ancestry, which alters the concept of type checking. Since descendent types inherit from their ancestors, a value of a descendent type may be safely used as a value for a variable that was declared to be of an ancestor type, since all properties of the ancestor are also present in the descendent. This conclusion is also valid for function parameters with mode "in," with the formal parameter type being an ancestor of the argument type. For "out" parameters, the type of the passed argument (which will receive the value) must be an ancestor of the formal parameter type, for the same reason. Finally, if the parameter's mode is "in-out," there must be an exact match between types.

5.3.5 Derived Types

In some languages, new types may be defined in terms of an existing type, but with the property that values of the new type are not acceptable where values of the original type are expected, and vice versa. In essence, this is an intentional creation of a new abstract data type that is equivalent (in the sense defined above) to the original type. This can be modeled as a type environment that has inherited the original type, but does not acknowledge the original type as an ancestor. Thus values of the derived type are not acceptable where values of the original type were expected.

5.3.6 Subprogram Types

By virtue of the fact that functions and procedures may contain declarations, they must include environments to contain these declarations. In the IAW, subprograms are modeled as type environments that are augmented with parameterization and return type information. Formal parameter declarations are placed both in the local declaration set and an ordered parameter list. The return type (known as a value type) is maintained as an instance variable. The type signature of a subprogram is constructed from the type signatures of its arguments and its value type. These type signatures are unique: two functions that have identical argument types and return types will have the same signature.

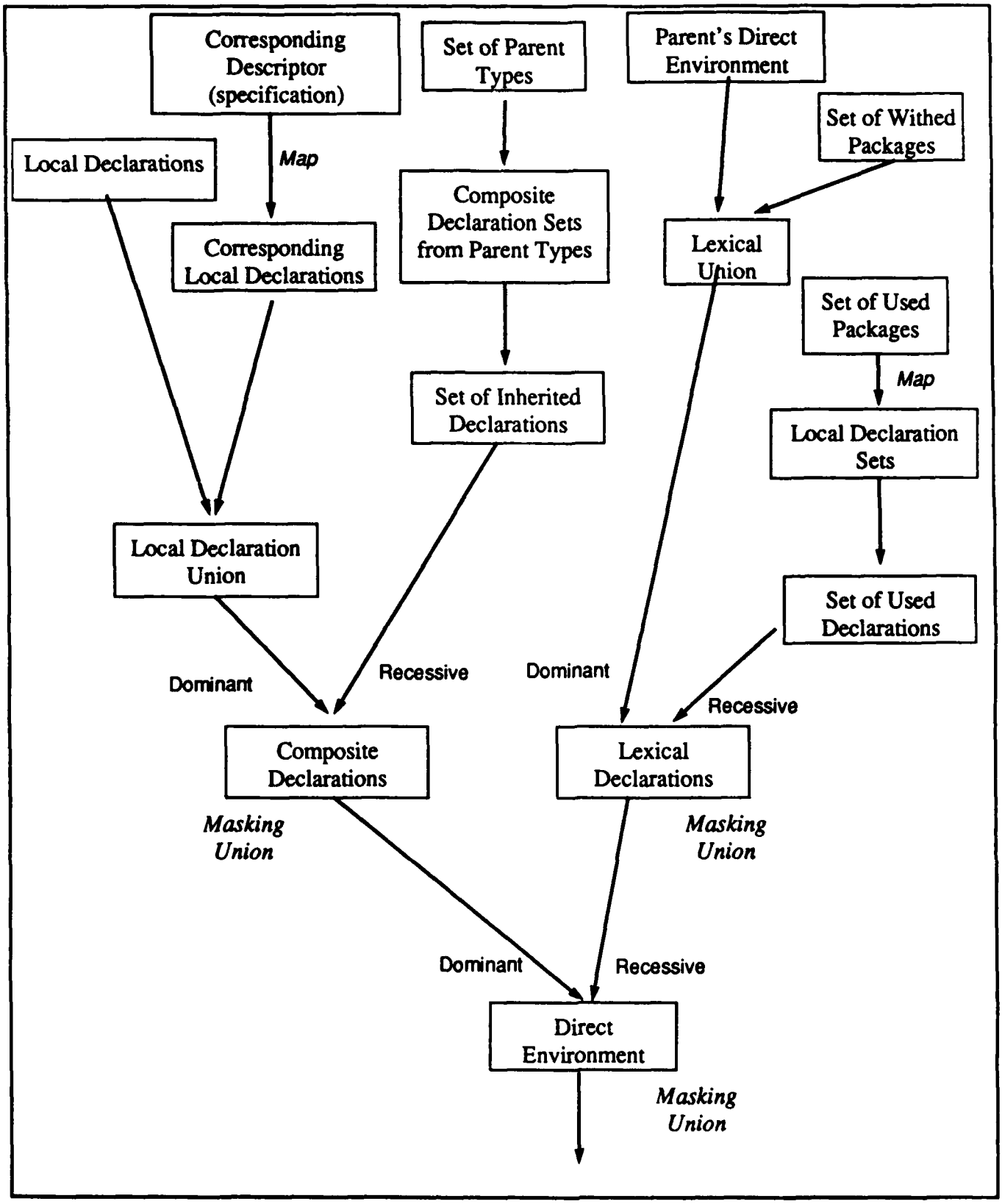


Figure 5-6. Full Ada Direct Environment Computational

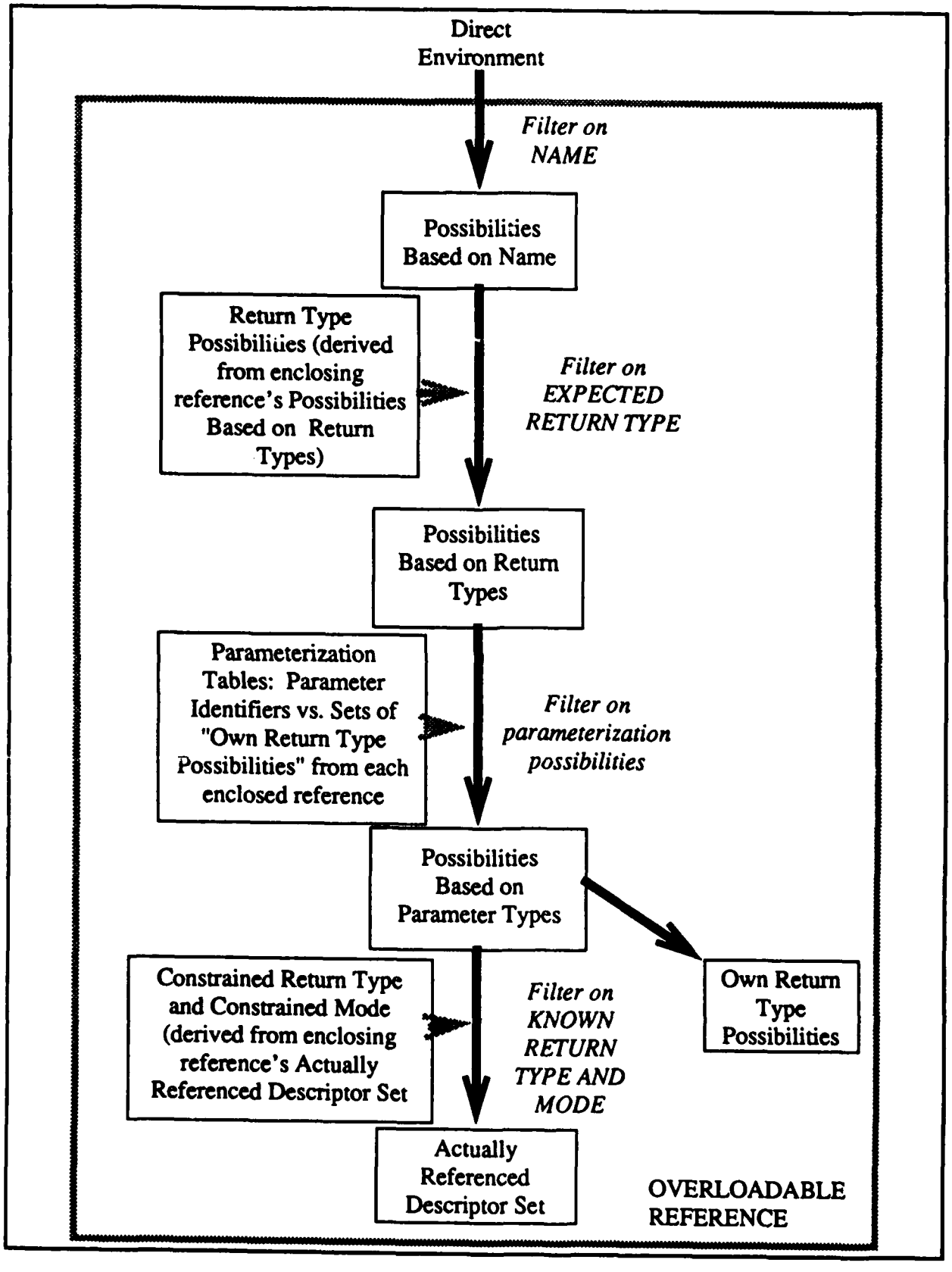


Figure 5-7. Overloadable Reference

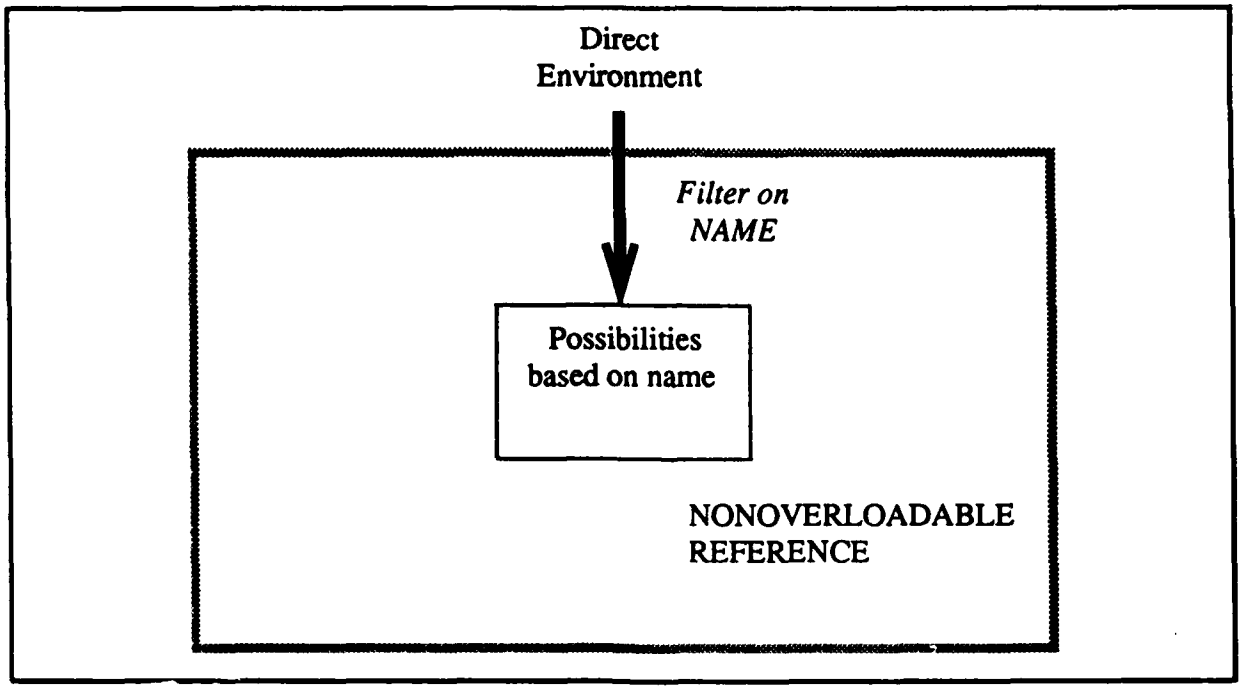


Figure 5-8. Nonoverloadable Reference

5.4 References

When an identifier is used in a language expression, the declaration to which the identifier refers must be determined. This correlation is, in actuality, a computation that seeks to map the identifier to a single declaration that is visible in the context in which the identifier appears. In Ada, all references can be resolved to specific declarations at compile time. In some languages, notably object oriented languages and languages with dynamic scoping, resolution may not be possible until run time. What can be done, however, is to implement a typing strategy that will allow a compile time check to ensure that the run time evaluation will always result in locating a declaration.

In the following section, we will examine overload resolution in Ada and its implementation in the IAW.

5.4.1 Ada References

Abstract data model references are objects that perform this computation. Every reference contains an actually-referenced-descriptor-set, and the computation is deemed successful if it terminates with exactly one descriptor in this set. Any other result indicates a semantic error.

5.4.1.1 Ada Overloadable References

Overloadable References, in addition to locating descriptors with the correct name in the direct environment, also embody the rules necessary for overload resolution. To implement overload resolution, the references are linked together according to the hierarchy of the expression in which they occur. As an example of this hierarchy, consider the assignment statement:

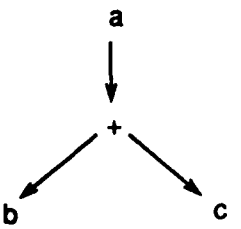
```
a := b + c;
```

In this statement, there are four identifiers: 'a', '+', 'b', and 'c'. For each of these identifiers, a reference is created to determine which declaration is to be associated with each identifier. The declarations need not be unique based on name alone. For example, there may be one '+' operator for integers, and another for real numbers. The references must determine which of these descriptors is the correct one.

In order to determine the correct descriptor, the references must share type information. To Help understand this process, let us reformulate the expression as if '+' were a conventional function:

```
a := + (b, c);
```

Obviously, the type of the value that '+' returns must match the type of a. In implementation, this is accomplished by the reference to 'a' providing type constraint information to the reference to '+'. The reference to 'a' is considered to be the *enclosing reference* for the reference to '+'. Similarly, the reference to '+' is the enclosing reference for the references to both 'a' and 'b', as indicated in the following diagram:



After this enclosing/enclosed reference relationship is established, the references share the type information necessary to uniquely select the referenced descriptors at each level in the reference hierarchy. As a side effect, references retain partial results that may aid the programmer if the reference did not uniquely resolve to a single declaration.

The process of sharing type information can be described in what is commonly known as the "three pass algorithm" for overload resolution:

PASS 1: The first pass propagates type possibilities from the outermost operator reference to the innermost. Given an expression organized as described above, the first pass of the algorithm begins with the outermost (topmost in the diagram) reference, and locates the declarations in the environment that match the identifier. This yields a set of declarations, from which the type possibilities of the next innermost reference may be derived. These type possibilities are then passed to the next innermost reference, which searches the environment for declarations that match its identifier and the type possibilities. This process repeats recursively to the innermost references.

PASS 2: The second pass proceeds from innermost expressions to outermost. Beginning with the innermost references, if a reference is a parameter of its enclosing reference, the set of descriptors found in pass 1 is used to determine the type possibilities that this parameter could assume. This type information is then passed to the enclosing reference, which examines all of the parameter type possibilities for all parameters, and further restricts its set of descriptors from pass 1 according to the parameter type information. If it, in turn, is a parameter of its enclosing reference, its type possibilities are in turn passed to its enclosing reference. This process continues until the outermost references have been processed.

PASS 3: The third pass is a final propagation of type information from the outermost to innermost references. After the first two passes, it is possible for all but the outermost references to still have multiple possibilities for referenced descriptors. If overload resolution is possible, the outermost reference must have only one possibility. The third pass consists of taking this one possibility and propagating type information once again as in the first pass, but operating on the sets of descriptors arrived at in pass 2. All references should uniquely resolve at this point.

If the references in the expression can be uniquely resolved, the outermost reference will be uniquely resolved after the second pass. The actual type information is then propagated to the enclosed references as in the first pass. All references should be uniquely resolved after the third pass.

As implemented, an overloadable reference locates a descriptor in a direct environment in four successive stages: locating descriptors with the correct name; locating descriptors whose return (base) types match the possibilities from an enclosing reference (pass 1), and providing constraints for enclosed references (actual parameters) as a result; locating descriptors whose parameter (base) types match the possibilities of the enclosed references (actual parameters) (pass 2); and, finally, selecting the single descriptor that is actually being referenced. More formally, the details of a single reference's Ada overload resolution can be described as follows:

Given:

Name n

Set of type constraints S_t (from the enclosing reference)

A final type constraint f

Set of descriptors D

Set of actual parameters A

The referenced descriptor may be computed by the following logic:

1) Locate descriptors with correct name

$$R_1 = \{d \in D \mid d'name = n\}$$

2) Further restrict based upon enclosing type constraints

$$R_2 = \{d \in R_1 \mid \exists\ t \in S_t \text{ s.t. } d'signature = t\}$$

3) Further restrict based upon actual parameter data

$$R_3 = \{d \in R_2 \mid \forall p \in d'parameters$$

EITHER

$$\exists a \in A \text{ s.t. } ((a'position = p'position) \vee (a'keyword = p.keyword))$$

$$\wedge\ p'signature = a'signature$$

OR

$$p'optional = \text{"true"}$$

4) Further restrict based upon the final type constraints

$$R_4 = \{d \in R_3 \mid d'signature = f\}$$

If R_4 does not contain exactly one descriptor, then the reference computation has failed, and there is a semantic error.

5.4.2 Extended References

The previous section described overload resolution in a context for which type comparison consisted of an identity check. When type inheritance is introduced, a variety of actual types may be acceptable for a given formal parameter type, depending upon the relationships of the types (in terms of inheritance) and the mode of the formal parameter.

Inheritance introduces a notion of ancestry. Formally, we define an ancestor of a type to be either the type itself, an immediate parent, or an ancestor of an immediate parent. Conversely, we can define a descendent type to be the type itself, a child of the type, or a descendent of any of its children. With these notions, we can now describe the type matching possibilities. For a formal parameter of mode "in," or a variable on the left hand side of an assignment statement, any value whose type is a descendent of the formal type is acceptable as an assigned value. For a formal parameter of mode "out," any value whose type is an ancestor of the formal type is acceptable as an assigned value. For a formal parameter of mode "in out," the types must match exactly.

In addition to this new notion of what types are acceptable as substitutes, an additional step is necessary in the overload resolution process. With the type ancestry comparisons instead of type identity, it is possible to reach the final stage of the old overload resolution approach and have more than one descriptor (this was a semantic error in the old method). When this occurs, the type of one of the descriptors in the set should be an ancestor of the types of all of the other descriptors. If it is, then this descriptor is the one being referenced. If such a descriptor is not found, then the reference cannot be resolved. In the following logic, a fifth step has been added to accomplish this check.

Extended overload resolution can be described as follows:

Given:

Name n
Set of mode and type constraint pairs S_t
Set of final mode and type constraint pairs F_t
Set of descriptors D
Set of actual parameters A

Define "appropriate-type" to mean "signature" if the reference has no parameters, and to mean "return-type" if the reference has parameters.

The referenced descriptor may be computed by the following logic:

1) Locate descriptors with correct name

$$R_1 = \{d \in D \mid d'name = n\}$$

2) Further restrict based upon enclosing type constraints

$R_2 = \{d \in R_1 \mid \exists\ s \in S_t$ s.t.

IF this reference is not a parameter of the enclosing reference
THEN s'type is an ancestor of d'appropriate-type
ELSE IF s'mode = "in"
THEN s'type is an ancestor of d'appropriate-type
ELSE IF s'mode = "out"
THEN d'appropriate-type is an ancestor of s'type
ELSE IF s'mode = "in-out"
THEN d'appropriate-type = s'type}

3) Further restrict based upon actual parameter data

$R_3 = \{d \in R_2 \mid \forall p \in$ d'parameters

EITHER $\exists a \in A$ s.t.
(a'position = p'position OR a'keyword = p.keyword)
AND
EITHER (p'mode = "in" AND p'signature is an ancestor of a'signature)
OR (p'mode = "out" AND a'signature is an ancestor of p'signature)
OR (p'mode = "in-out" AND a'signature = p'signature)
OR
p'optional = "true"

4) Further restrict based upon enclosing type constraints after parameter restriction

$R_4 = \{d \in R_3 \mid \exists\ t \in F_t$ s.t.

IF this reference is not a parameter of the enclosing reference
THEN t'type is an ancestor of d'appropriate-type
ELSE IF t'mode = "in"
THEN t'type is an ancestor of d'appropriate-type
ELSE IF t'mode = "out"
THEN d'appropriate-type is an ancestor of t'type
ELSE IF t'mode = "in-out"
THEN d'appropriate-type = t'type}

5) Finally locate SUP (least upper bound) signature descriptor

$R5 = \{d \in R_4 \mid \forall d_x \in R_4$ d'signature is an ancestor of d_x'signature}

If R_5 does not contain exactly one descriptor, then the reference computation has failed, and there is a semantic error.

6

Graphical Structure Editor and the Abstract Model

Summary

This chapter discusses the graphical editors' methodology, including the evolving representations as well as concurrent update and the intended level of representation.

It concludes with a section covering areas for further research.

6.1 Evolving Representations

The initial BRAT editor and its successor, the structure editor, had their own internal representations of a program that was independent of the language model. In the P3, P4 and L5 prototypes, a program was provided that created the graphics data structure from the language abstract model. Coupled with the ability of the graphics editors to generate Ada code, which could then be read into the language editor, this provided a primitive means of going back and forth between the two styles of editors. (Note that no mechanism has been implemented for going from the language view to the representations for the graphical behavior editors - there is still much research to be done in this area.)

The mechanisms provided in the P3, P4 and L5 prototypes for transitioning from graphics to language and back again have one serious drawback: they lose information. In going from graphics to text, the specifics of graphic placement and layout have no textual representations, and are lost. A sequence of transformations from graphics to text and back to graphics will thus result in a graphic image that is equivalent to the original image in the information that it contains about the program, but may look very different.

In going from text to graphics, information is also lost: there is no representation in the graphic structure editor for executable statements. A sequence of transformations from text to graphics and back to text will, therefore, result in text without any executable statements.

6.1.1 Concurrent Update

The goal of concurrent update is to eliminate this problem of lost information in going back and forth between the graphics and language views of the program. This work was in progress at the time that the stop work order was received. The following description relates the state of the design at that time.

The approach taken in achieving concurrent update was to rethink the structure editor as a view of the abstract model. Each descriptor in the abstract model would have one or more corresponding graphic views or images, each indexed by name. Since declarative unit descriptors may contain other declarations (these declarations are locally defined in the declarative unit descriptor), the views associated with these descriptors may, optionally, include the views of contained descriptors. The resulting information model is shown in figure 6-1.

While the mapping between descriptors and views is straightforward, the relationship between references in the code and arrows (invocations) in the graphics is a little more complicated. The interpretation that has been given to the arrows in the graphics is that they do not represent any specific reference that may occur in the code. Rather, they represent a constraint that at least one reference to the indicated declaration must occur within the scope of the indicated descriptor. The information model showing this relationship is shown in figure 6-2.

6.2 Intended Level of Representation

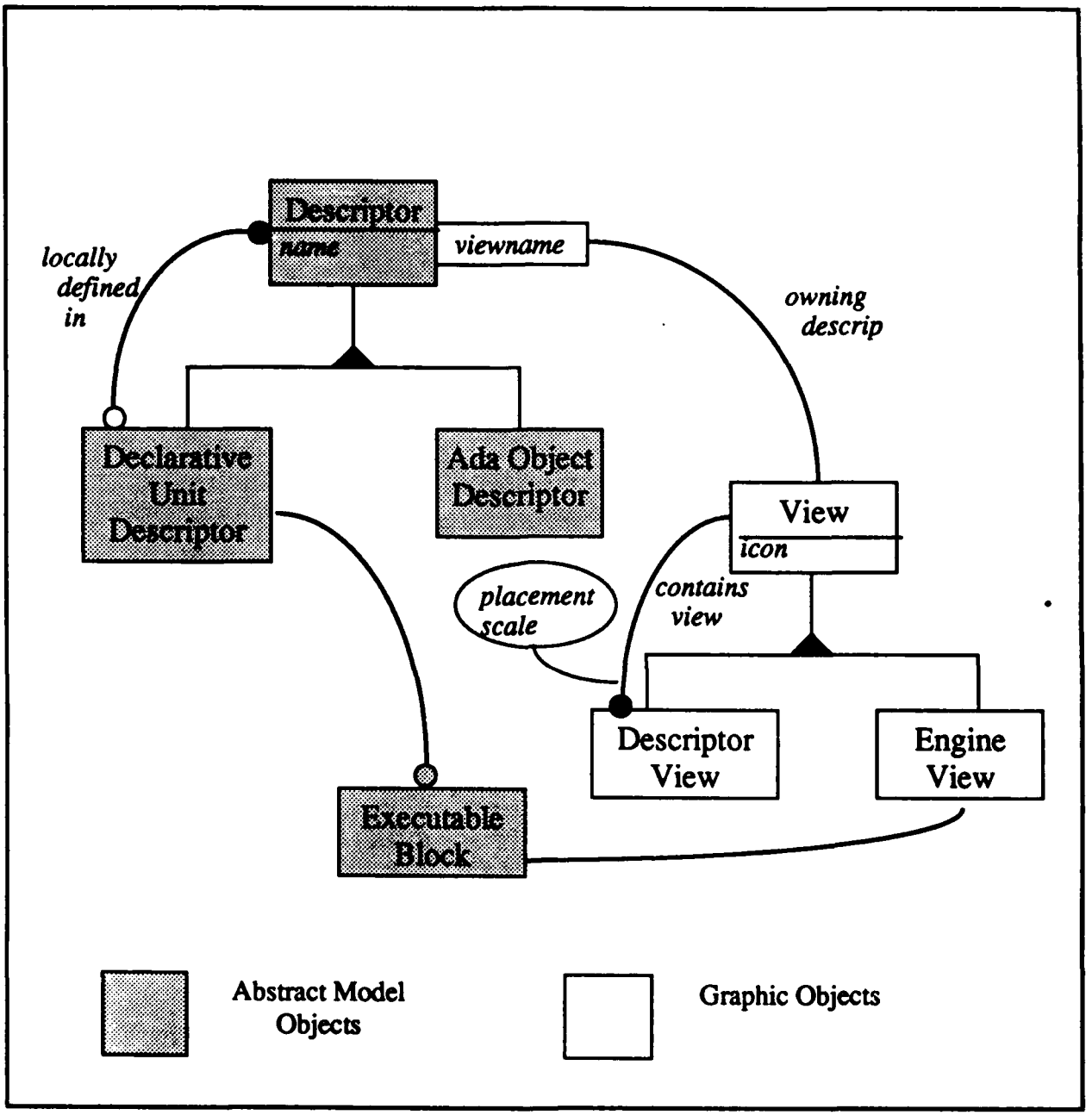


Figure 6-1. Structure Editor Views of the Abstract Model

☞ *Consistency requirement: If view V1 contains view V2, then V2's owning descriptor must be defined in V1's owning descriptor.*

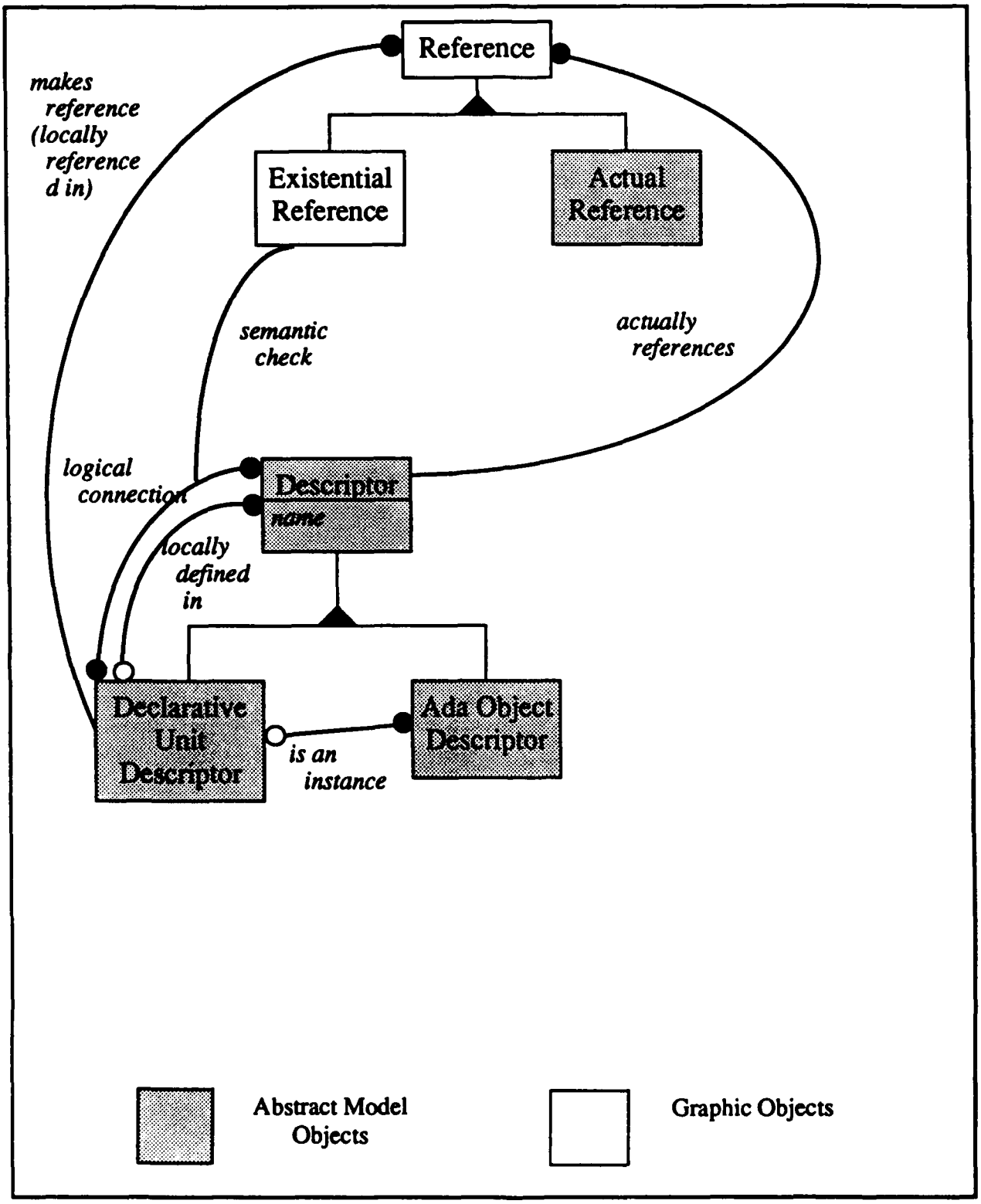


Figure 6-2. References

6.3 Areas for Further Research

One major area that remains to be examined is the relationship between the graphic representations of behavior and their manifestations in code. While it is relatively simple to generate code that implements a state machine or decision table, that code representation is not unique: there are many possible programs that implement the behavior defined by a single state machine. On the other hand, it is not possible, in general, to determine if a particular piece of code has a state machine representation, and recover that representation. Thus we do not even have the rudimentary relationship between text and graphics that we had for the structure editors.

How, then, do we maintain the consistency of the graphic and textual views? Part of the answer seems to be that the abstract representation of a state machine may, in fact, be substantially different than that of the code that is generated from it. In this case, the projected code could be treated like a noneditable artifact of the abstract state machine. Numerous issues remain to be explored, such as:

- What is an appropriate abstract representation of behavior?
- How should a mixed representation of "real" text and noneditable text be presented to the user, and managed from a language (parsing) point of view?

7

Language View Interaction with the Abstract Model

Summary

This chapter provides a functional overview of the language processing part of the system, including diagrams of the P1 and P2 prototypes, the P3, P4, and L5 prototypes, and the P6 prototype, which is under development.

It also contains a description of the abstract model interface.

7.1 Functional Overview

The language processing portion of the system has evolved considerably over the course of the IAW contract. Figure 7-1 is a data flow diagram of the initial language processing implementation, as delivered in the P1 and P2 language systems. Note that all communications between the editor and the displayer (except for cursor placement information, which is not shown) occurs through the parser. Note also that the parse tree is a data structure that is only used by the parser.

In figure 7-2, this architecture has been extended to include attributes on the parse tree, and the processing necessary to compute their values. Also, the abstract model and its associated processor have been added. The attribute processor computes the values of attributes on the tree based upon other attribute values. Some of these computations have side-effects of modifying the abstract model and generating semantic error messages.

In the P3, P4 and L5 prototypes, the abstract model processor updates the attributed parse tree to provide feedback about the results of semantic analysis, and the displayer takes on the additional responsibility for the annunciation of semantic errors. Thus there are three processes that can access the attributed parse tree: the parser, the attribute processor, and the abstract model processor. This situation requires a high degree of access coordination of these three processes. While some rearchitecture of the control structures was done in this series of prototypes to improve responsiveness to the user, this continuing interaction made testing and troubleshooting difficult.

The architecture for the (uncompleted) P6 prototype was altered as reflected in figure 7-3 to overcome this coordination problem. In this arrangement, the responsibility for the annunciation of semantic error messages has been moved to a separate tool. In consequence, the information flow from the attribute processor to the abstract model processor, and, ultimately, the error tool is strictly one-way. This completely eliminates the three-way coordination problem found in the earlier prototypes.

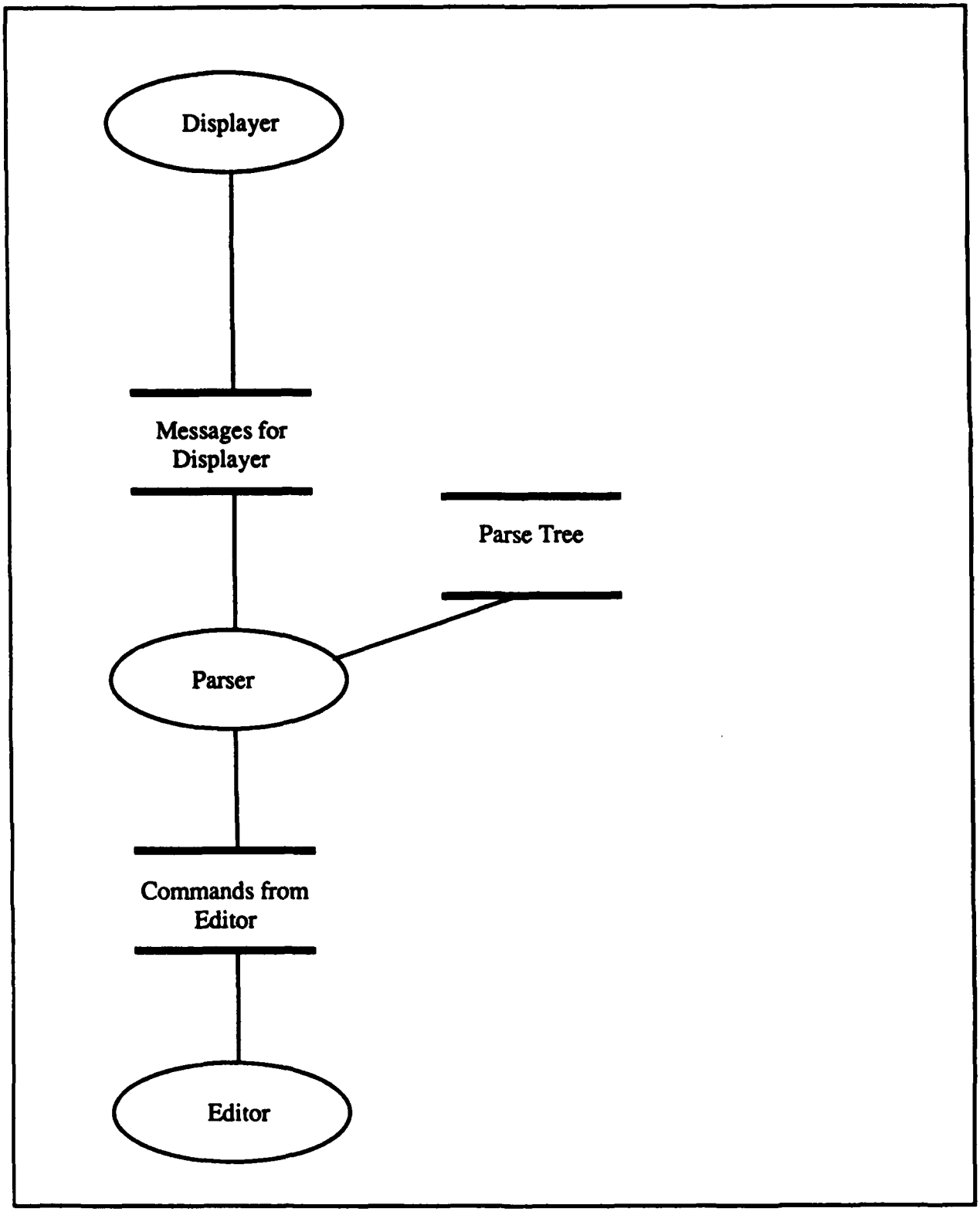


Figure 7-1. P1 and P2 Prototypes

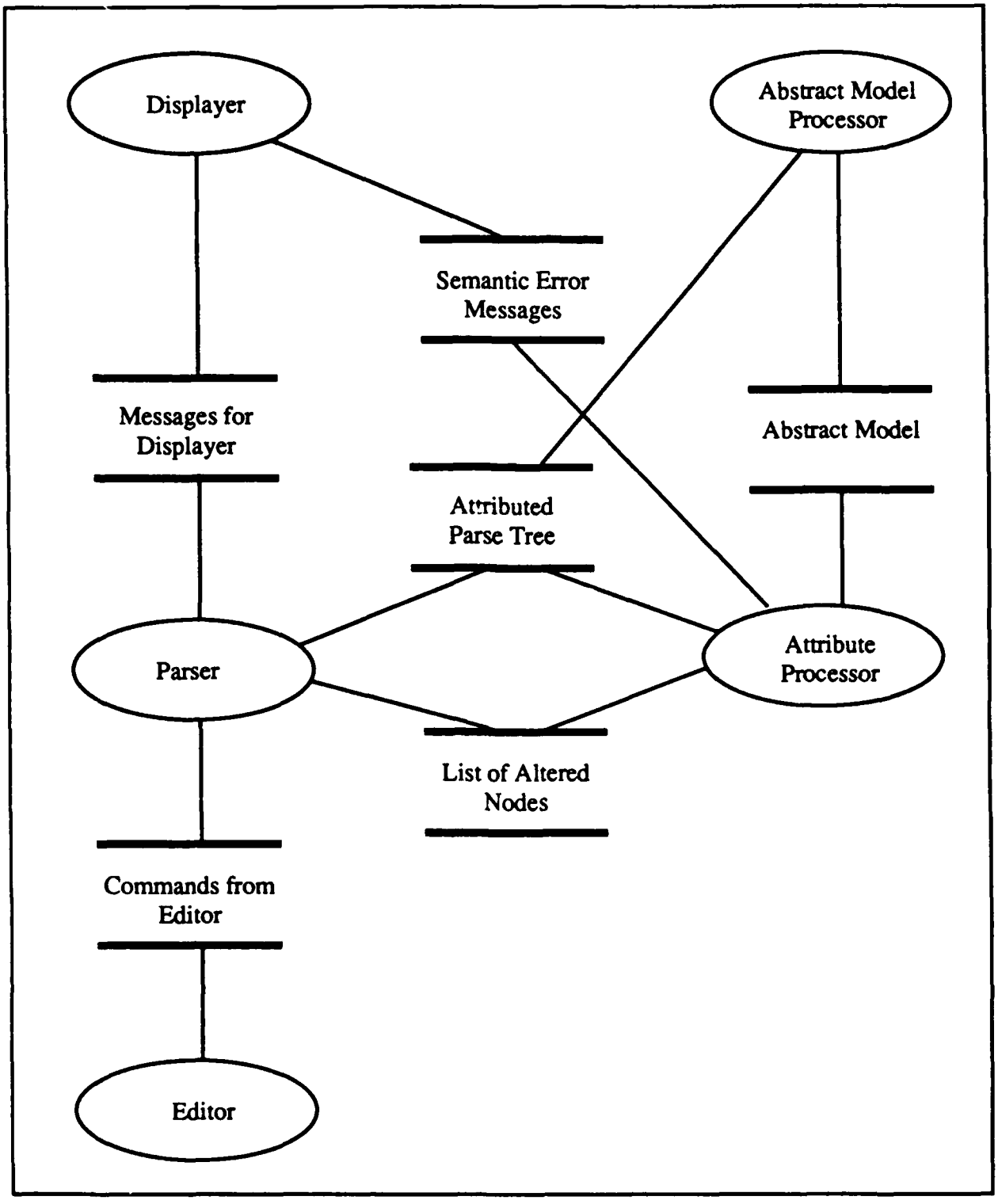


Figure 7-2. P3, P4 and L5 Prototypes

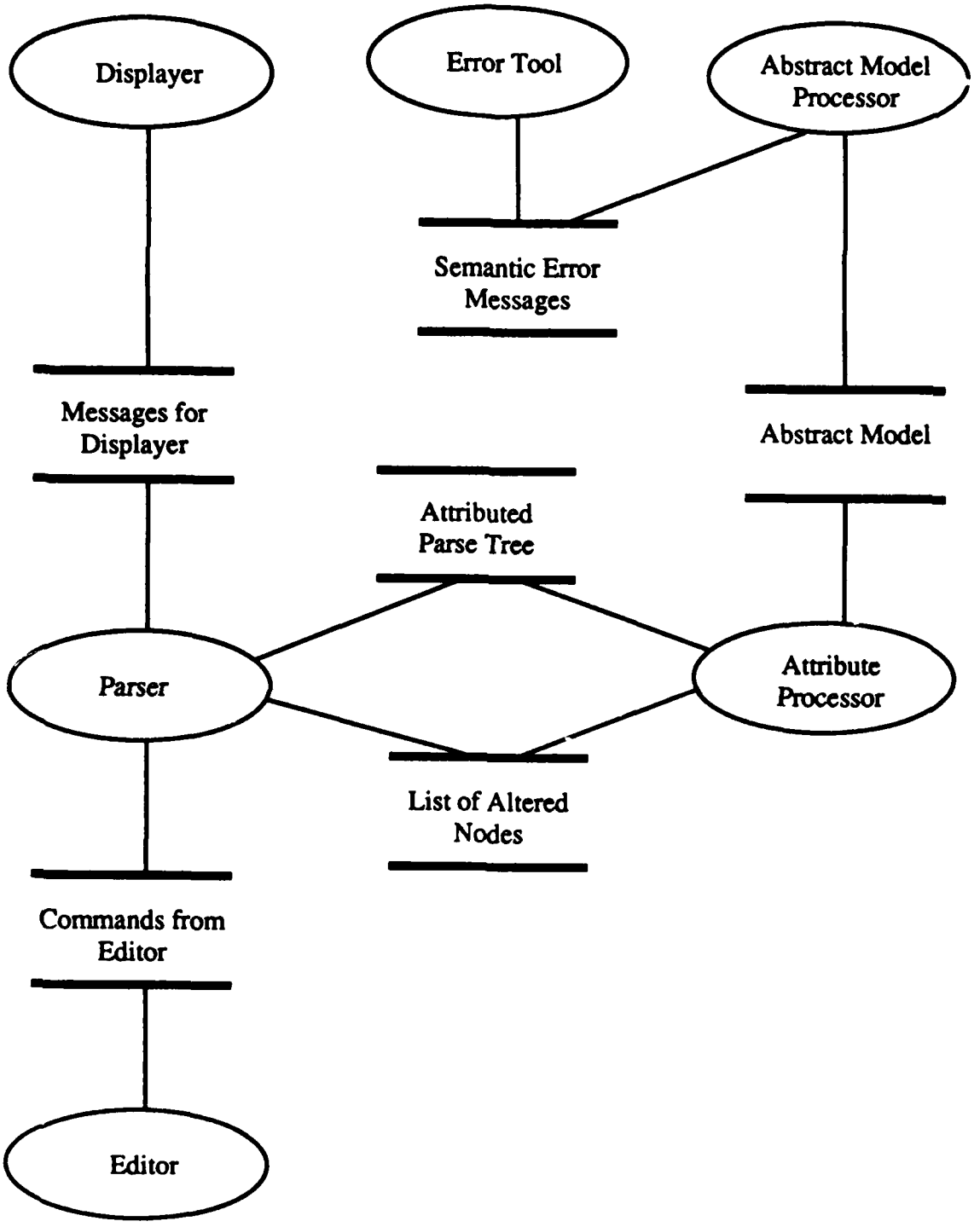


Figure 7-3. P6 Prototype (under development)

7.2 Abstract Model Interface

The nature of the interface between the parser and the abstract model has also evolved from prototype to prototype. The basic elements of the abstract model are descriptors (representatives of program elements that have been declared) and references (identifiers that need to be matched up with the declarations that they are referring to). In the P3, P4 and L5 prototypes, the attribute processor explicitly created both descriptors and references, as well as initiating semantic checking on these objects.

One problem with this approach is that, with some type expressions, it is not clear whether the expression simply refers to an existing type, or is the implicit declaration of a new subtype. To overcome this problem, special abstract model functions were introduced to construct descriptors. These functions evaluate the type expressions, and decide if a descriptor needs to be constructed, or whether an existing descriptor is being referenced.

With this change, the attribute processor no longer creates desciptors at all. All interactions now fall into one of three categories:

1. the creation and destruction of abstract semantic model references and sets,
2. the modification of parameters of the references,
3. and the placement and removal of abstract semantic model objects in sets.

8

Interpreter (Abstract Model)

Summary

The IAW Interpreter is composed of three parts:

1. the inspector,
2. the kernel, and
3. the error-handler.

The inspector is a tool to hierarchically examine the structure of the program definition or its execution.

The kernel is the primary component and allows a user to interactively execute a program.

If a trace point or error is encountered, the error-handler provides a mechanism to trap and display the current state.

8.1 Abstract Model Interface

8.1.1 Interpreter Information

This section describes the information provided by the Interpreter for the user. Three classes of information are available through the interpreter:

1. Program Source Information
2. Program Execution Information
3. Interpreter Control Information

These classes of information are provided through the three components of the Interpreter. Together, they give a complete picture of the definition and execution of a specified program. To understand the information, some characteristics of the interpreter should be understood. The interpreter executes an Ada program in subunits corresponding to Ada tasks. A default task is assigned to the overall program, and a program may or may not include user-defined tasks.

8.1.1.1 Program Source Information

Program source information shows the correspondence of execution information with the program structure and program text. The following sections describe the program source information that is maintained.

8.1.1.1.1 Source Display

The source code display of a unique editor is associated with the interpreter. As a result, the code will be displayed using boldface type and other pretty printing techniques. The kernel will add information to the editor window as needed.

8.1.1.1.2 Program Structure

One capacity of the inspector is displaying the program structure. The inspector allows the user to look closely at a single program object, e.g., an if statement, a variable, or a package. The inspector displays detailed information about the object chosen. The user can then choose to look closely at another program object. This object may be an item in the previous objects display, e.g., a procedure might be chosen from the local declaration information about a package. In this way, the user could examine the program structure in a hierarchical manner. The user could also examine unrelated pieces of the structure as needed. The inspector operation is described in detail in section 8.1.2.

8.1.1.1.3 Source Trace Point Indicator

For each trace point, a marker pointing to a line in the source display is maintained. If a trace point is set for a statement, the marker indicates that the trace point action is associated with that statement. For a trace point set on a variable, the indicator will be placed on the line of the variable declaration.

8.1.1.1.4 Execution Pointer

For each activated task the kernel has a source code execution pointer. This pointer tracks the line of source code corresponding to the task's execution point. If viewing the entire program, the display will include only the execution pointer of the task in which the current execution is taking place. If viewing the execution of a single task, the execution pointer of this task is the one displayed.

8.1.1.2 Program Execution Information

Program execution information is the largest class of information associated with the interpreter. As the program runs, information is stored and retrieved. It is this information that determines the behavior of the execution of the next line of code and allows the user to track this behavior. The following is a breakdown of execution information.

8.1.1.2.1 Task List

A list of all activated tasks is maintained. Associated with each task is an execution state that is one of *RUNNING*, *READY*, *SUSPENDED*, and *TERMINATED*. [ANSI 83]

8.1.1.2.2 Data Stack

A data stack contains all of the current runtime data for a task. The data is grouped according to the declarative unit to which is belongs, i.e., all of the variables declared in Function A and its parameters are grouped separately from the parameters and variables declared in Procedure B. Each activated task has a private data stack. These stacks are linked through a global data stack. This global stack also stores all of the runtime data for packages and other units that do not belong in an Ada task.

8.1.1.2.3 Control Stacks

Like the data stack, a control stack for each task is maintained. The control stack is a stack of all of the program units and statements that have been called but not yet completed. For example, if Function A calls Procedure B, which consists solely of Loop C, the control stack at the beginning of execution of Loop C will consist of Function A, Procedure B, and Loop C. As execution of Loop C proceeds, the control stack may also include instructions contained within Loop C. As a program unit or statement is completed, it is removed from the control stack, i.e., after completion of Loop C, the control stack consists of Function A and Procedure B.

8.1.1.2.4 Execution History

The interpreter is designed so that it remembers all of the steps a program takes during execution. The history is a trace of exactly what the interpreter has done with any associated data changes and is maintained on both a global and per task basis. Space (memory) concerns may require that some elements of the history, typically *executable units* such as loop statements and procedure calls, get compressed into one atomic action. With this trace of the execution history, it now becomes possible to unexecute the program either on the global or per task basis. The execution of an individual task can be unexecuted without affecting the status of other tasks until a point of rendezvous is reached. To back up beyond this point, each task at the rendezvous must be unexecuted.

8.1.1.3 Interpreter Control Information

The following information describes the state of the kernel. It allows the user to adjust the behavior of the interpreter and determine the status of the program or interpretation.

8.1.1.3.1 Trace Points

Trace points are set, activated, and removed by the user. Before and after executing each statement or accessing variables, the kernel checks for a trace point. If a trace point is reached, user defined actions will occur, such as removing the trace point and continuing, or entering the error-handler. The error-handler allows examination of the status of the interpretation more closely.

The interpreter allows flexibility in this debugging environment by providing extensive trace point functionality. Trace points can be set on Ada statements to take action before or after the line is executed. Breaking before the line is executed is the traditional use of breakpoints. In addition, trace points can be set on variable declarations. In this case, the user specifies whether a break will occur each time the variable is accessed, written, or both. This is the traditional use of watch points.

User-modified attributes allow these two types of trace points to be selectively activated and modified individually or in groups. The following is a list of attribute descriptions. Each attribute includes a list of options in braces with the attribute title and its default setting in bold.

- **Action to Take at Beginning**: {**Option_Menu**, Error, Nothing, [Halt, Deactivate, Print, Remove]}

  The specified action will be performed before the statement at which the trace point is set is executed or before the variable on which the trace point is set is referenced.

  The user can choose one of the first three options above and can choose one or more of the options in brackets. *Option_Menu* displays a menu of the other action options at the time the trace point is hit. *Error* puts the current execution state in the error-handler. *Nothing* simply tells the interpreter to continue execution.

One or more of the following options can be chosen. *Halt* halts execution completely. *Deactivate* will cause the breakpoint to become inactive so further execution will not break at that point. *Print* indicates the source line/data is printed in the Debug window pane. *Remove* removes the trace point and all associated indicators.

- **Action to Take at End: {Option_Menu,** Error, Nothing, [Halt, Deactivate, Print, Remove]}

  The specified action will be performed after the execution of the line at which the trace point or after referencing the variable on which the trace point is set. The options are the same as described in the attribute *Action to Take at Beginning*.

- **Reason to Break: {Read, Write,** Value}

  This attribute indicates the conditions to be satisfied when breaking on a reference to a variable. A variable is only associated with a trace point set on a declaration; therefore, the *Reason to Break* attribute is not applicable to trace points set on Ada statements.

  One or more of the options can be chosen. *Read* causes the specified action to be performed if the variable is accessed for reading. *Write* causes the specified action to be performed if the variable is accessed for writing. *Value* allows the user to qualify the trace point to only be active if the variable is equal to a user inputted value.

- **Direction: {Forward, Unexecution}**

  Trace points have a direction associated with them. The direction tells the error-handler to pay attention to the trace point when interpreting in that direction. One or both of the options can be chosen.

- **Active: {Yes,** No}

  This parameter indicates whether or not the trace point should be effective immediately. This attribute can be used when one wishes to set multiple trace points and then invoke them one at a time. Note also that trace point classes may be a more efficient way to control the activation of trace points.

- **Class:** {**Null,** Class_Name}

  Each trace point belongs to a class. The advantage to using classes is that sets of trace points can be activated and deactivated as a group. This can be more efficient than setting the *Active* attribute for each individual trace point. The default indicates that the trace point is a member of an unnamed class. Class names are assigned by the user when trace points are created. Any valid Ada identifier can be used for a class name.

8.1.1.3.2 Selected Tasks

Task selection controls the display of portions of the symbol table and the history information. The user selects one or more tasks of a program. The symbol table shows global data and the data owned by the selected task or tasks. The displayed history shows only what the selected task or tasks has done. The default selected task is the set of all tasks of the current program.

8.1.1.3.3 Execution Direction

The kernel is able to execute an Ada program in both a forward and reverse direction. The program executes normally in the forward direction. The reverse direction unexecutes the code. Commands have been defined to allow the direction of successive execution steps to be independent.

8.1.1.3.4 Current Stepping Size

The size of the step to be executed next is determined by the command issued. A micro-step executes one line of assembly code. A step executes one line of code. The user can also choose to execute a statement or a loop iteration. The interpreter also allows finishing the execution of an indentation level or a call from the current execution point.

8.1.1.3.5 Debug Package

The kernel provides a package for debugging that is separate from normal program I/O. The package allows statements to be flagged as debug statements. The debug statements can be executed or ignored. Ignoring these trace statements might be useful in checking the performance of the code once it has been debugged.

The output from the debug package can be displayed in two ways.

- To retain the context, debug output can be interleaved with standard program output.
- To separate the trace statements, a separate window can be used for display purposes.

The user will also be able to save and print the debug package interaction.

8.1.2 Inspector

The inspector component of the interpreter allows the examination of the program structure and execution. When an object is inspected, detailed information about that object is displayed. If an Ada package specification is inspected, the information would include all variables and subprograms defined in the package specification. From this point, the user can inspect any object displayed as information from a previous inspection or choose something unrelated.

8.1.2.1 Window

The inspector uses a form of the typical interpreter window and is shown in Figure 8-1. Commands are found in the four menus on the menu bar. There are three panes that hold inspected objects and one pane for a history of the inspected objects. The inspection history provides an easy method of calling back information that has already been examined. The display of the first inspected object will appear in the bottom pane of the window. Its information will be presented in a tabular form with labels. The object name will appear in the inspection history. As a new object is inspected, the earlier inspections are entered into the inspection history and filter up a window until they drop off the top. To recall an item that has already been inspected, the mouse can be used to pick an object from the inspection history or from the inspection panes. All panes of the inspector window are scrollable to hold more information.

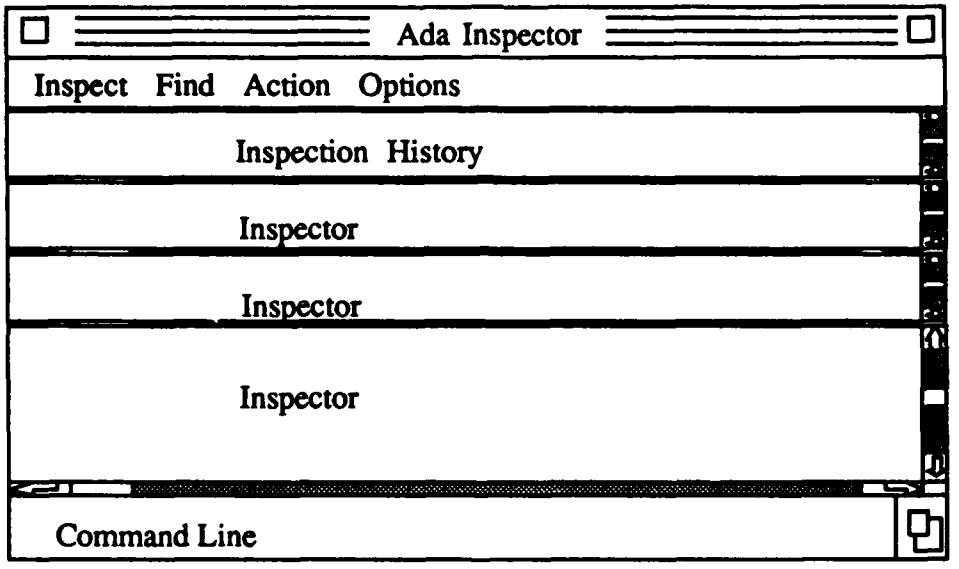


Figure 8-1. Inspector Window

8.1.2.2 Inspector Operation

A sample inspector session better illustrates the inspector operation and its capability to examine a structure hierarchically. The user may, for example, first specify Package AAA to examine the compile-time structure. The inspector would produce information about this package including any tasks that are declared locally as shown in Figure 8-2. Note the listing of information includes Task One, Task Two, and Type State. Package AAA has been entered in the inspection history.

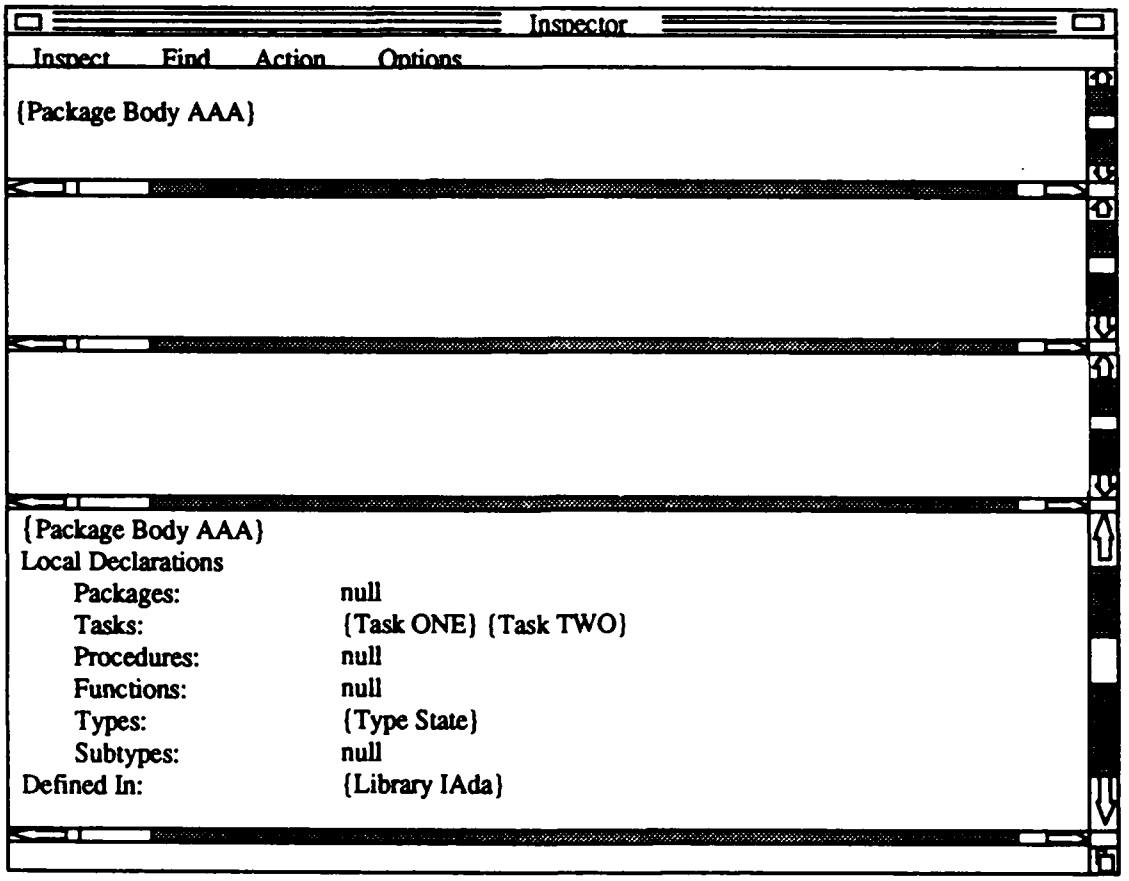


Figure 8-2. Inspection of Package Body AAA

The user may from there, inspect Task ONE which was declared inside of Package AAA. In Figure 8-3, you can see the inspector at this point. This is inspection in a hierarchical manner. Package AAA has been examined. When Task ONE was examined, Package AAA moved up one inspector pane. The information displayed depends on the type of the object under inspection. Both entries are now in the inspection history.

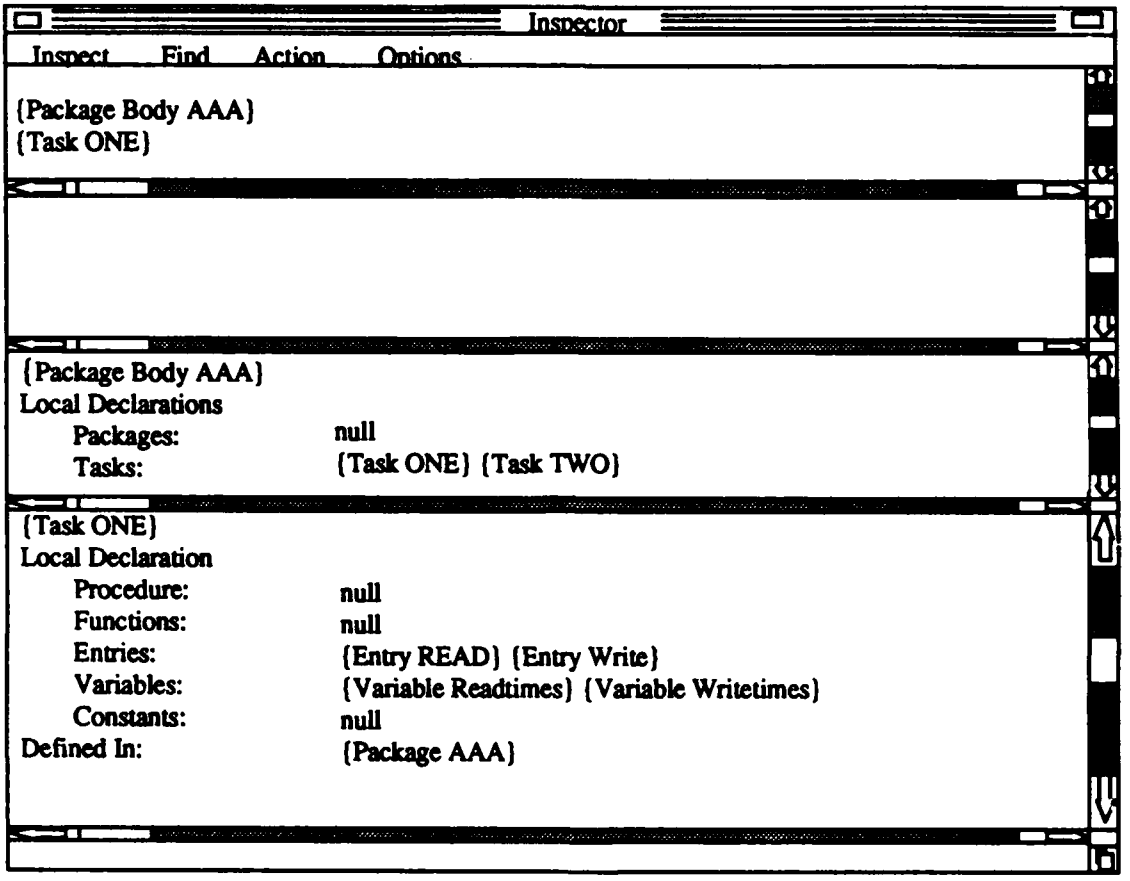


Figure 8-3. Inspection of Task ONE

To examine the program execution, the execution executive might first be examined. The information displayed for the inspection of the runtime structure is vastly different than that displayed for the compile-time structure. From there, an activated task may be inspected.

Inside this task, a history stack element or a control stack element might be examined. The resulting display can be seen in Figure 8-4. It is a hierarchical inspection of the execution state.

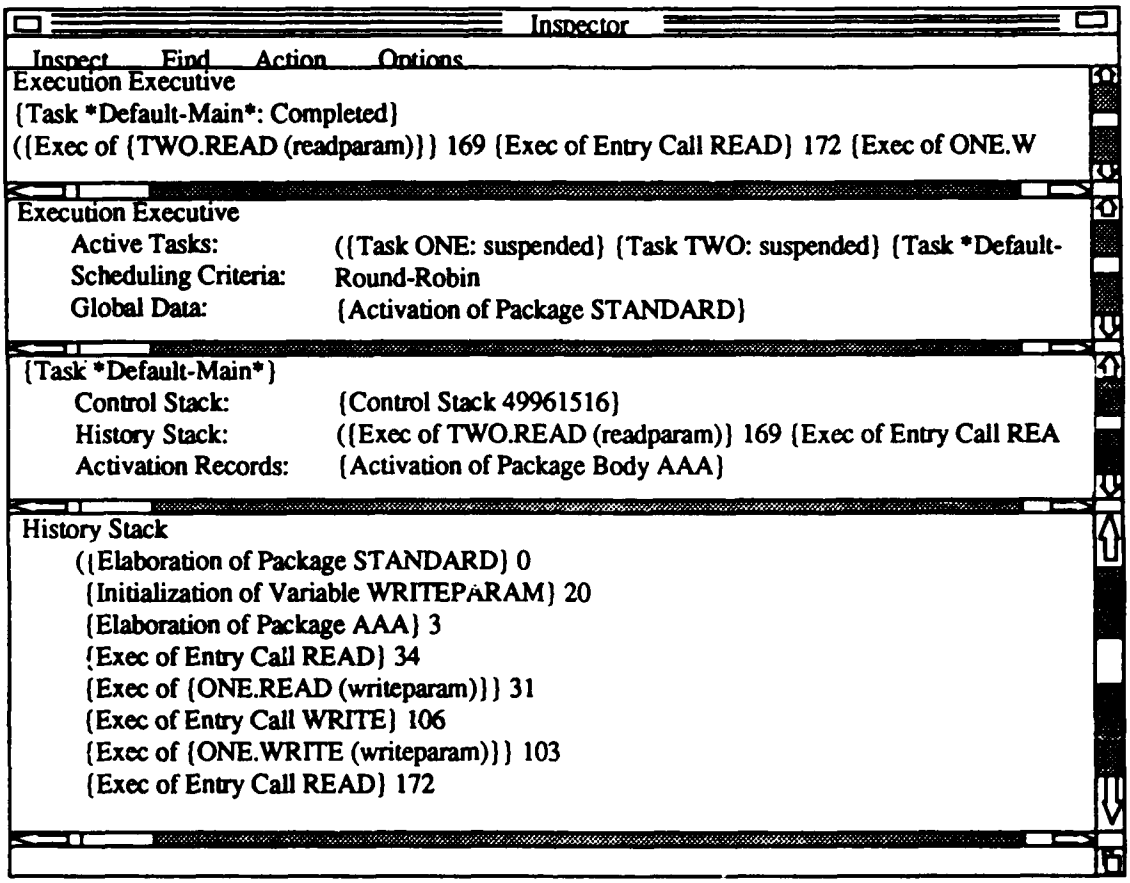


Figure 8-4. Inspection of Runtime Structure

8.1.2.3 Inspector Commands

The user can issue commands in the inspector to inspect or find certain objects. There are several input methods for the commands. All commands available through the menu bar at the top of the window are also available through the command line at the bottom of the window. Some commands can be transparently invoked by a mouse action or key binding. Additional commands are initiated only by the mouse. The mouse is especially useful in the inspector to indicate which object should be inspected next.

The commands have been divided into pull-down menus accessed through the menu bar and mouse activated commands. Each of the categories of commands will be introduced and the associated commands discussed in detail. If an input menu is used for several commands, the

listing of each of these commands will follow the input menu description and indicate it is contained on that menu. These commands are also individually accessed through the Command Line. Figure 8-5 displays the commands available and the menus to which they belong.

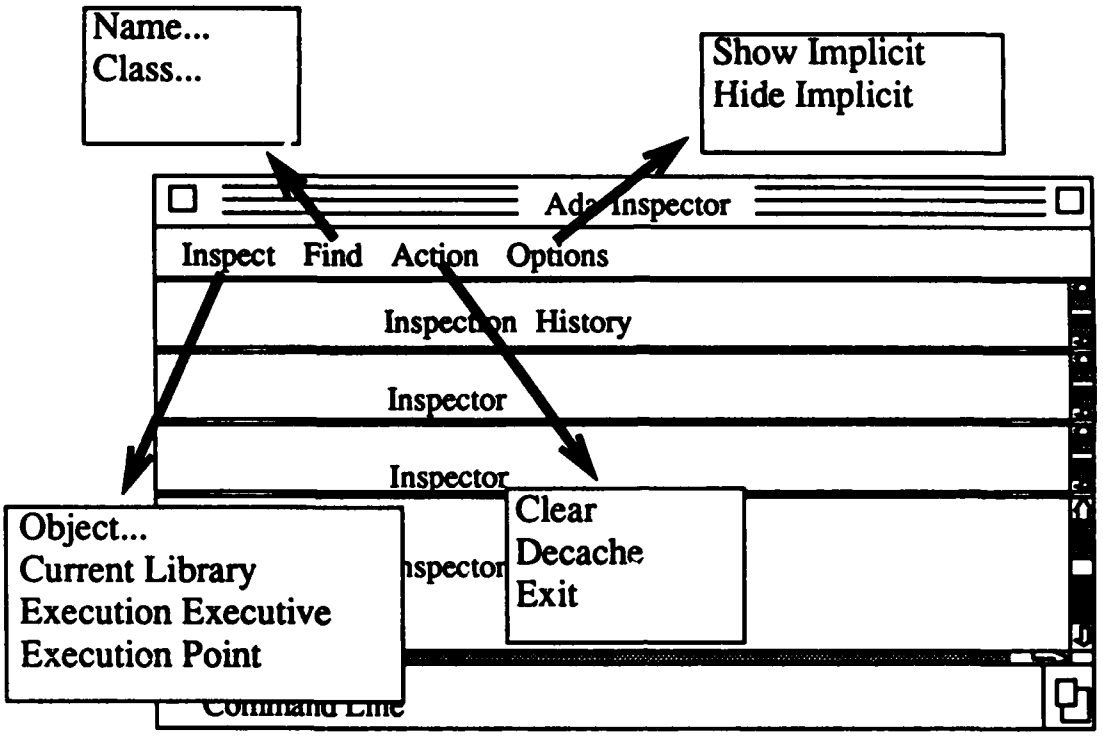


Figure 8-5. Inspector Commands

8.1.2.3.1 Command Template

For each command, a listing will appear in the format shown in Figure 8-6. Each listing will include first the command title and any arguments. A default key binding will be given if applicable and an indication if the command is available through the mouse. Next, a description of the command and its arguments will be given. The listing will include a description about the input method, if necessary. This is needed, for example, if an intermediate window appears for additional input when using the menus. Last, if there is any default it will be noted.

Command Title **{argument 1}** **default key binding**
{argument n} **mouse action**

Discussion of command functionality and interface. This description might include an illustration of an input frame needed to receive input. If there is a default for this command it will be explained here.

Figure 8-6. Command Listing Template

8.1.2.3.2 Inspect

The next object for inspection is specified with the commands in the Inspect Menu. The object then appears in the lowest inspection pane, and the object identifier appears at the bottom of the inspection history pane.

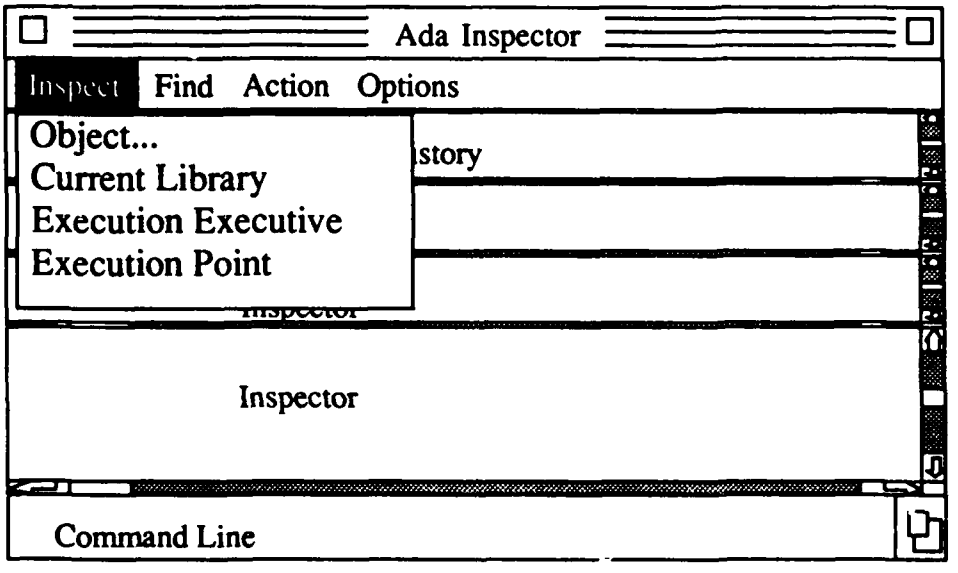


Figure 8-7. Inspect Menu

Object... **{identifier}** **no default key binding**
no mouse action

A frame appears to receive the identifier that specifies the object to be inspected.

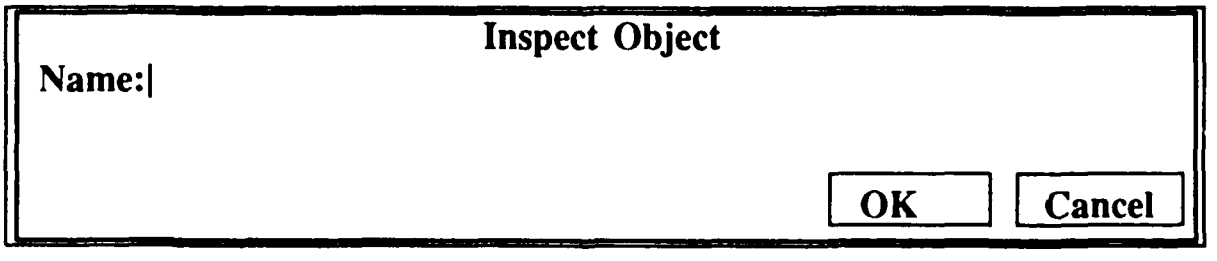


Figure 8-8. Inspect Object Input Frame

Current Library **no default key binding**
no mouse action

The top level compile-time structure of the current library is inspected. This is the package that implicitly contains all compilation units declared for the current library.

Execution Executive — **no default key binding**, **no mouse action**

The top level of the execution state is inspected. This is an inspection of the execution of the code, not the structure of the code. The information displayed includes the set of activated tasks, the global history stack, and trace point table.

Execution Point — **no default key binding**, **no mouse action**

The execution state at the current execution point is inspected. Unlike the inspection of the *Execution Executive*, inspecting the *Execution Point* is the examination of a single activated task. Information about this task includes its control stack, private history stack, and run state.

8.1.2.3.3 Find

The Find Menu allows a user to inspect an object without knowing its complete unique identifier. An argument is given for each command which directs the inspector to look for certain objects. A list is then returned as the object to inspect. From there, the mouse can be used to inspect a specific element of this list of objects.

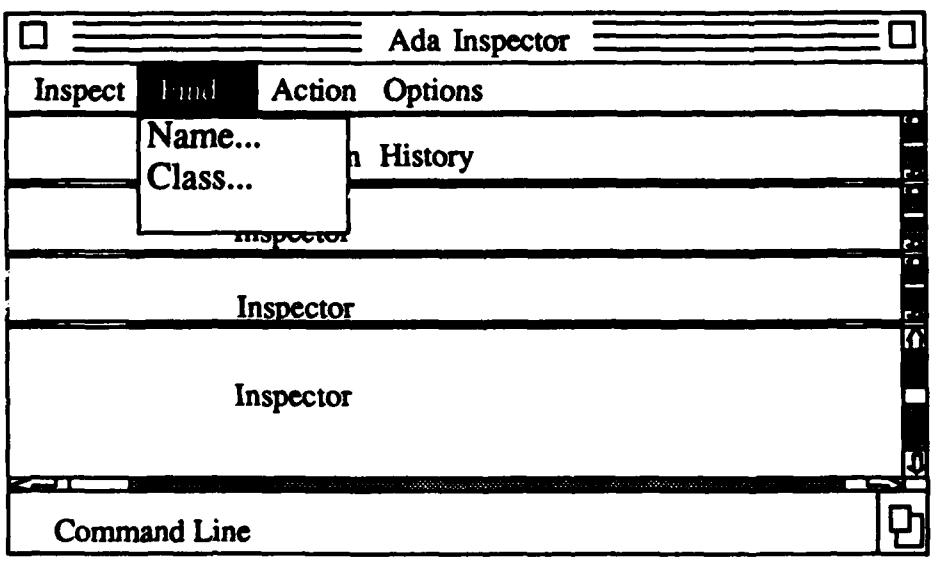


Figure 8-9. Find Menu

Name... — **{identifier}** — **no default key binding**, **no mouse action**

This command accepts an identifier as an argument. The compilation unit is then searched for each occurrence of this identifier. A list of the objects

named by the identifier is returned to the user. The frame used for input is displayed in Figure 8-10.

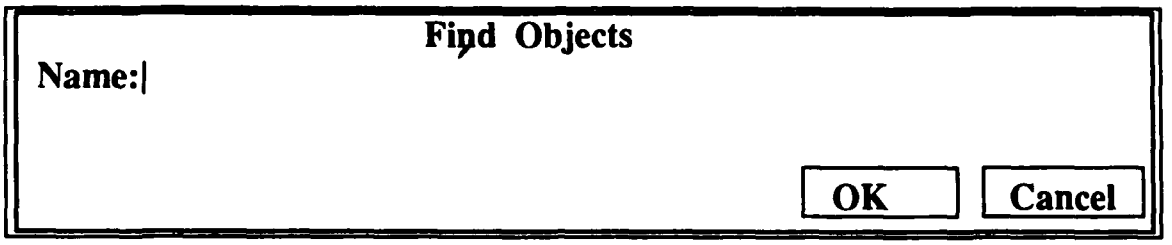


Figure 8-10. Find Name Input Frame

Class... **{class of object}** **no default key binding**
no mouse action

One or more classes of object such as package, variable, or operator are chosen as search qualifiers from the menu as it appears in Figure 8-11. A list is then returned with all objects of that class. The search can, but need not, be restricted by entering a specific name. If executed from the menu, the input frame contains the classes available. The *All Program Units* option is available to easily choose all classes of program units with one action.

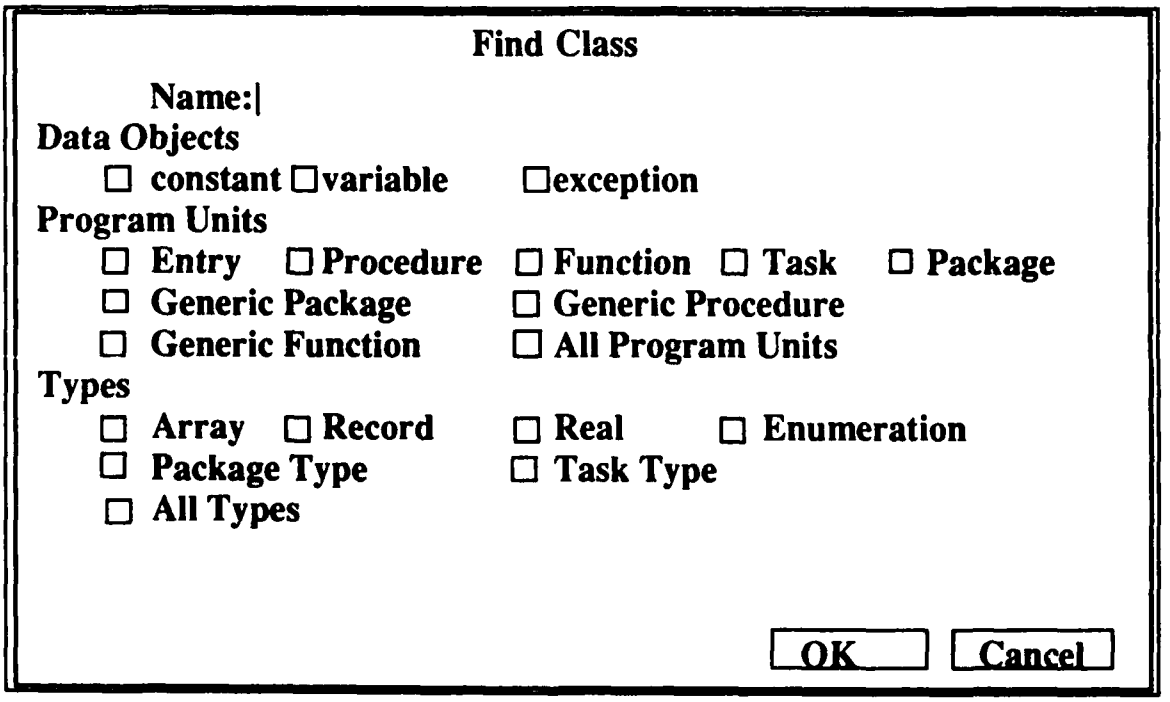


Figure 8-11. Find Class Input Frame

8.1.2.3.4 Action

The Action Menu contains commands that act on all panes of the inspector window.

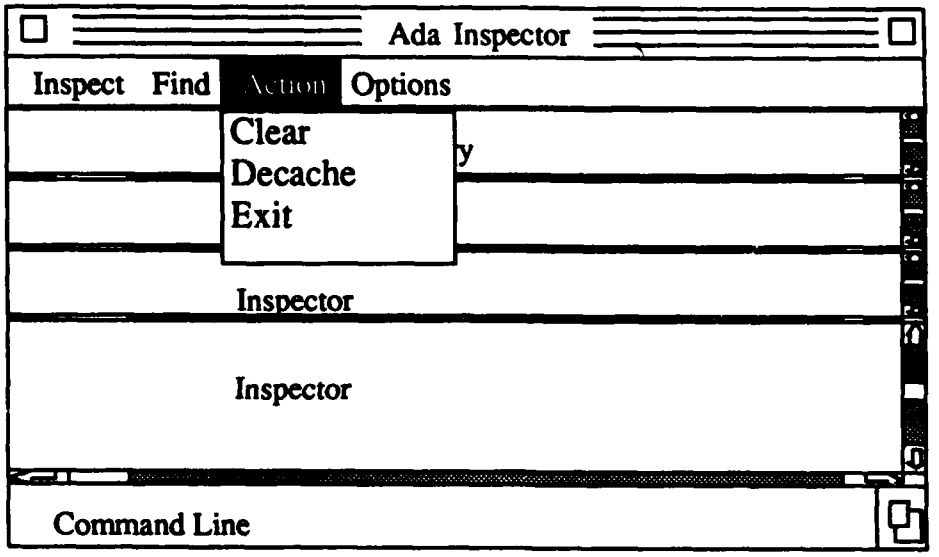


Figure 8-12. Action Menu

Clear **no default key binding**
no mouse action

The display of the inspector windows and the inspection history are blanked.

Decache **no default key binding**
no mouse action

This command is a mechanism to ensure that any changes made to the source of the program will be reflected in the inspection of the program. It is a good idea to *Decache* when returning to the inspector after interpreting or editing.

Exit **no default key binding**
no mouse action

This command quits the inspector, returning to the caller.

8.1.2.3.5 Options

The commands of the Options Menu allow some specific customization of the inspector information.

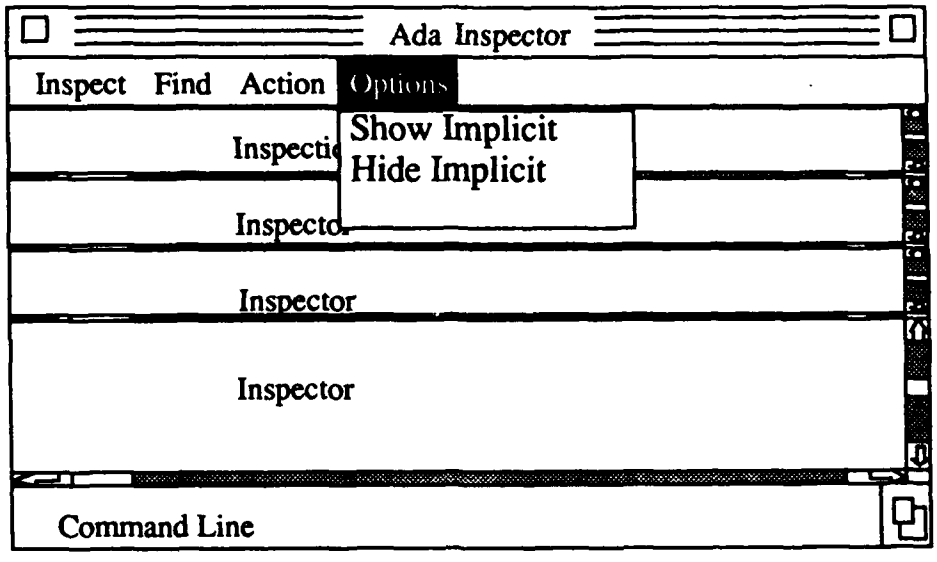


Figure 8-13. Option Menu

Show Implicit **no default key binding**
no mouse action

All the implicit program structure as well as the user-defined program structure is displayed. Implicit information such as the function '+' could appear in an inspection pane. The inspector is defaulted to show implicit information.

Hide Implicit **no default key binding**
no mouse action

Implicit information is hidden from the inspector display. This allows the user to more easily scan the user-defined compile-time structure. Upon examination of all functions, only user defined functions will appear. Functions such as '+', IN, or AND will not be displayed. The inspector is defaulted to show implicit information.

8.2 Abstract Semantic Model

Each object in the Abstract Model is tagged with a field which indicates whether it is semantically correct and, therefore, interpretable.

9

Porting Considerations

Summary

This chapter explains porting considerations, including:

- Portability to New Platforms
- Strongly Typed vs Weakly Typed Languages

9.1 Portability to New Platforms

Several issues arise in considering the portability of the IAW to different platforms. Perhaps the dominant issue is that of language, which is dealt with in the next section. The remaining considerations are the nature of the operating system support for process and interprocess communications, and the facilities available for windows and window management.

9.1.1 Operating System Process Support

In the P1 and P2 language systems, the Editor/Displayer and the Parser are in separate processes. These processes communicate frequently, at least once for every keystroke. In the P6 prototype, the abstract model becomes yet a third process, with additional communications between the parser and the abstract model, and between the graphics structure editor and the abstract model.

This frequent interprocess communications requires that the operating system provide efficient support for context switching and a scheduler capable of working in small time increments. A possible alternative would be the operating system support of lightweight processes.

9.1.2 Window Systems

The largest obstacle to porting the IAW is the window system. Despite the creation of a generic window system as part of the project, this window system is built upon the Symbolics window system, which has many capabilities not found in other windowing systems. The language editor on the Symbolics utilizes the Symbolics "scroll maintain list" capability to manage interaction with the mouse in several of its windows. Capability similar to this is not found in any of the major window systems in use. The IAW Inspector is built on top of a substrate used for the Symbolics inspector. In order to port the Inspector this substrate would all need to be reengineered and coded. In all likelihood, most of the window dependent portions of the IAW would have to be completely recoded to make them work under another window system.

9.2 Language Issues

The IAW was written in Lisp with the Flavors object-oriented extensions. The simplest means of porting would appear to be to take the Lisp code and simply execute it in the desired workstation environment. Aside from window and operating system issues (addressed in the previous section) this approach would require workstations with configurations substantially larger than minimal (24MB physical memory) to get good performance.

Alternative approaches require the translation of the Lisp code into another language. C with an object oriented extension (C++ for example) is a likely candidate. Translation into languages like Ada and Modula-2 that do not have object oriented extensions would probably require a major redesign.

10

Productivity Gain Estimate

Summary

This chapter relates productivity issues, including:

- Examples
    - Productivity Gains - Structure Editor
    - Productivity Gains - State Machine Editor

10.1 Preliminary Results

Due to the complexity of the language processing work, none of the delivered prototypes were complete enough in language coverage to be used in an actual project in order to derive productivity data. However, some preliminary data is available from the use of two of the graphic editors. These results are reported in the following sections.

10.1.1 Productivity Gains - Structure Editor

Several trial experiments were run in an effort to appreciate the productivity gains that are possible using a graphic structure editor. In these experiments, one person was asked to develop a Buhr diagram of a piece of software using pencil and paper, and then use an ordinary text editor to create the code outline implied by the diagram. A second person used the IAW structure editor and the code generation capability to perform the same tasks. In these experiments, the structure editor approach showed a 5:1 to 10:1 productivity improvement.

We must realize that these results only cover a small portion of the development life cycle. The code generated is simply an outline of the program units indicated by the diagram, and all of the actual executable code still has to be added. The effort required for this portion of the design process was not included in the experiment.

10.1.2 Productivity Gains - State Machine Editor

One of the research programs at the GE Corporate Research and Development Center expressed an interest in using the state machine editor to design communications protocols. This organization spent a considerable amount of time evaluating the state machine editor, and reported back productivity improvements ranging from 5:1 while experimenting with protocols up to 50:1 while formally developing final protocols.

The circumstances surrounding this rather astounding result must be understood in order to interpret the productivity gains correctly. First, it must be recognized that, in communications protocol design, once the state machine is specified, the job is done. This is in sharp contrast to most other state machine applications, in which action routines (specifying what to do when a particular state transition is taken) must be added to the state machine in order to get a complete program. The second point that must be recognized is that this organization actually had two groups of engineers: communications engineers, who create the state diagrams; and software engineers, who translate the diagrams into executable code. In this particular application, the use of the state machine editor allowed the complete removal of the software engineer from the process.

The research program that performed these experiments is now supporting the conversion of this editor to "C" on a more readily available workstation platform so that it can become part of a communications engineer's development environment.

11

Contract/Accomplishments Comparison

Summary

This chapter describes the overall project accomplishments including: prototypes, the core system (IAda language, interpreter, hot editor, generic window interface, and project Data Base), the Smart Librarian, productivity measurements, additional tools, the help system and expert system tools.

11.1 Prototypes

As discussed in earlier sections, the original contract called for the production and delivery of seven prototypes. Due to changes in the total amount funded, and alterations in the schedule of disbursement, only five prototypes were actually delivered under the contract. A sixth prototype, which was developed with GE funding during a hiatus in the contract funding, was delivered to the Air Force under the terms of a memorandum of understanding regarding proprietary information. A seventh prototype, which would have been delivered under the contract, was under development at the time that the stop work order was received.

11.2 Core System

The core system consists of the IAda language, the Interpreter, the Hot Editor, the Generic window Interface, and the Project Data Base. The following sections describe the accomplishments in these areas.

11.2.1 IAda language

The initial IAda language (referred to as the P1 language) was a simple expression language that was used to demonstrate the functionality that was expected from the editor, interpreter and hot editor. It was delivered with the P2 prototype. The P2 language began to approximate true Ada. It relaxed constraints on declaration ordering, and allowed bodies to be entered before their specifications. The syntax of the language was nearly identical with Ada, differing only in that some constraints that are enforced syntactically in the Language Reference Manual become semantic checks in IAda. As the prototypes progressed from P2 through P4, L5, and the P6 prototype that was under development when the stop work order was received, more of these constraints were moved from syntax to semantics. For example, in the latest version, an empty sequence of statements is allowed anywhere that statements are expected. This allows the program to be analyzed in other ways, and simply generates a warning that this is not legal Ada.

The P2 prototype contained a parser (syntax analyzer) for the entire Ada language (with some relaxation of checking). P3 introduced semantic checking for simple types, packages, tasks and subprograms, as well as assignment statements. P4 included attributes for simple types, user defined operators, and case statement support. The L5 prototype continued the expansion of type coverage, and work was in progress toward full language coverage at the time that the stop work order was received.

11.2.2 Interpreter

The interpreter design was begun during the development of the P2 prototype. In that prototype, the initial form of the abstract semantic model (then called the filter-net, which was part of the IFORM) was defined. In P3 and P4, the parser extensions necessary to construct portions of the abstract model from the text were put into place, and the interpreter itself was added. L5 extended the language coverage of the interpreter still further, and work was in progress toward full language coverage at the time that the stop work order was received.

11.2.3 Hot Editor

The hot editor, as designed, is not really a separate entity, but rather a mode of operation of the system. The intent was to provide a capability of interpreting a program while making changes to the program, and to have this be a single interactive process from the user's perspective.

The initial design of the system, in which the parser serves to recognize program constructs and build their abstract representations in the abstract model, and the interpreter operates from the abstract model itself, lends itself well to this concept. From the P3/P4 prototypes on, changes to the text that did not affect the current point of execution would be gracefully accommodated by the abstract model and the interpreter.

The interpreter was designed to have an execution history record, which could be used to undo the execution of the program to just before the faulty statement, so that the code could be corrected and execution resumed. This capability was being implemented when the stop work order was received. Also in progress was the implementation of a scheme to recognize when changes to the program being executed affect regions that have already been executed, so that the point to which execution must be backed up can be automatically recognized.

One further level of hot editor behavior that was also being worked on was concurrent update: the ability to alter the program being executed from the graphics window as well as the language window.

11.2.4 Generic Window Interface

During the development of the graphics editors, a number of standardized capabilities were implemented for menu and form presentation and management, as well as icon and connection drawing and object selection. Progress on standardizing this aspect of the system was hindered substantially by four major revisions to the Symbolics operating system that occurred during the course of the contract.

11.2.5 Project Data Base

The abstract model was designed so that the individual compilation units of the abstract model could be stored and retrieved independently. When the funding was terminated, work was in progress to not only implement this capability, but to save the state of the semantic analysis as well. This capability included versioning of the individual compilation units.

11.3 Smart Librarian

The smart librarian portion of the project focused on constructing a tool that would allow the user to browse and select elements of a library for inclusion in his design. An initial prototype of the librarian was delivered with the P4 prototype.

This tool basically took a directory containing the library elements, and an additional file that gave rules for querying the user for information necessary to select the appropriate library element. The initial library consisted of the Booch data structures, as described in his book.

Having gone through the exercise of implementing this librarian, we recognized that the real challenge was not in implementing the tool, but in developing a categorization and location scheme that was appropriate for the library. We felt that further research into these issues would not contribute significantly to the improvement in productivity of a programmer.

11.4 Productivity Measurements

Due to the lengthy development time of the language processing portion of the system, the IAW never matured enough to be used on a project level. Some data were obtained on the use of the graphic editors alone, as described in chapter 10.

11.5 Additional Tools

The BRAT editor, state machine editor, and decision table editors evolved substantially through the P2, P3 and P4 prototypes. Non-boolean capability was added to the state machine editor, and Brat editor was evolved into a structure editor capable of showing both hardware and software designs.

Stand-alone data flow and tree editors were developed and delivered as part of the P2 prototype. The tree editor was used extensively during the development of the incremental parser. For the P3 prototype, the dataflow editor was coupled to the Brat editor to allow designs started with data flow analysis to be transitioned into BRAT where they could be completed.

When the project was terminated, work was under way to modify the structure editor to use the abstract model as its internal representation. This work is described in more detail in chapter 6.

11.6 Help System

A elementary version of the help system was delivered with the P0 prototype. This system allowed users to browse the user's manual on-line. For the P1 prototype AI techniques were added to the help system allowing the user instant access to information concerning the operation that they were performing. A more advanced version of the help system that incorporated command scenarios was developed after P1 was delivered. This allowed the user to see exactly how a command worked by watching the system as it performed the command on a sample database.

11.7 Expert System Tools

Using the Delphi expert system, several tools were developed to aid in the system design process. In the P1 prototype, tools were delivered that guided users in the choice of data structures and in the selection of searching and sorting algorithms. These systems were designed to ask users a maximum of ten questions and then come up with a recommendation. If users had questions about the recommendation, the system was able to provide them with the reasoning used in making the selection. Two more expert system tools were added for the P2 prototype; these were the task communications assistant and the mode type assistant. These system were also built using the delphi expert system.

12

Lessons Learned

Overview

This chapter contains an overall project evaluation and reflects upon some of the lessons learned throughout the development process.

It includes some discussion of rapid prototyping methodology, incremental compiler technology and of multiple views built upon a central database.

12.1 Project Evaluation

As mentioned elsewhere in this report, the development of the language processing technology turned out to be a more formidable task than expected. At the outset, there was a conscious decision that the development of a compiler would not be attempted as part of this project. However, during the course of the project, what amounts to a new compiler technology had to be developed to provide the necessary functionality for the various tools.

The central importance of the language technology to the overall operation of the workstation forced a redirection of resources into the language work, at the expense of some of the more peripheral items such as help systems and expert advisors. Had the language technology existed at the start of the project, the full scope of planned work would have been achievable within the planned scope of effort.

12.2 Rapid Prototyping Methodology

Rapid prototyping makes one fundamental assumption concerning the development: that the results of one step may be evolved into the next prototype. In the case of the IAW language technology, this did not turn out to be a valid assumption. The P1 language prototype was developed using a subset of Ada for which some simplifying assumptions could be made. These simplifying assumptions made it easy to develop the suite of tools (editor, analysis, interpreter), but since those assumptions were not valid for the entire Ada language, all of the tools had to be completely redone over the next three prototypes.

Looking at the rapid prototyping process with a 6-month cycle, a typical breakdown of work over the 6-months is as follows:

- Month 1: Cleanup from the previous prototype. Bug fixes, support, analysis of lessons learned, direction setting for the next prototype.
- Month 2: Analysis and design for the next prototype. Decide in detail what is to be done. Allocate resources for remainder of prototype.
- Months 3&4: Coding. Implement the design. Unit test the components.
- Month 5: Integration and integration testing. Ensure that the essential functionality is in place and working.
- Month 6: Stabilize the prototype and Document. Make sure that the annoying little "crash" bugs are found and removed. Provide user documentation. Prepare for the review.

In retrospect, it appears that rapid prototyping may be appropriate for the development of highly interactive systems that do not have large computational tasks embedded within them (such as semantic analysis), but its use in an application that has such a large computational task at its core is questionable. This is particularly so when the nature of the computation or the method of implementing it is not well defined at the outset. The use of the rapid prototyping paradigm in this situation skews the focus of the work away from spending the time to analyze the computational task in favor of trying to deliver something that is demonstrable as the next prototype. In the short rapid prototyping scheme, there is not enough time to do significant analysis and modify the design based on the analysis within the 6-month cycle.

12.3 Incremental Compiler Technology

While the extension of the semantic analysis to the full Ada language is not yet complete, it appears that the notion of adding an incremental compiler to an editor is quite feasible given a 2-4 MIP workstation with 8-12 MB of physical memory for each user. For the interpreter to be useful on larger programs, extensions to the approach would have to be made to allow the use of compiled code in conjunction with the interpreted code. Operating in this mode, the modules under test would be interpreted, and compiled code would be used for the other modules. A preliminary investigation indicates that this is practical, but would probably require the active participation of the compiler vendor to implement.

12.4 Multiple Views Built on a Central Database

Achieving this goal was a major objective for the final prototype that was under development when the funding was terminated. While the work was not completed, enough progress was made to make some observations.

The most difficult task in reaching the goal of multiple views of a single database is to select the appropriate concepts to model in the database and to interpret each feature of the text and graphics for this model. For some aspects of a design, the representation and mapping is obvious: a declaration in the language view is mapped into an abstract representation of the thing being declared; similarly a package in a structure diagram can be readily mapped into its abstract representation, and, indirectly, related to the language representation.

A subtle distinction arises when we consider the meaning of a function call in the language view vs. an arrow from the caller to the called function in the graphics view. If we attempt to equate the two, then there must be an arrow in the graphics view for each occurrence of the function call in the language view. Furthermore, the presence of the arrow in the graphic view does not provide enough information to recreate the language view: it does not specify where the reference occurs in the code view, nor does it specify the relative placement of references.

Our consideration of this distinction has led us to interpret that the arrow in the graphics view represents a constraint that there must be at least one reference to the function in the language view. The implication here is that the reference in the language view is, on an abstract level, different than the arrow in the graphics view, and thus has a different abstract representation. Consistency checking between these abstractions now becomes part of the abstract model processing.

To consider the graphical behavior editors, such as the state machine and decision table editors, a more difficult problem arises: there is no unique mapping from the language view to the graphics view, and back again. And, the behavior described by a given state machine may be implemented by any number of programs. Conversely, it is an open question as to whether a given segment of code is, in fact, a state machine. How to deal with this situation remains an open question.

Index

U

V

W

☆U.S.Government Printing Office:1990—748-056/24507